A Spectrum of IV&V Modeling Techniques
NAG-11971

Deliverable

Title: Adapt the Suite of 6 Flight Guidance Models

WBS/Task: 4

Date: June 9, 2003

Grant

Number: NAG-11971

Project Title: A Spectrum of IV&V Modeling Techniques

Contractor: University of Minnesota

Principal Investigator

Name: Dr. Mats P.E. Heimdahl

Title: Associate Professor

Phone: (612) 625-2068

Fax: (612) 625-0572

Email: heimdahl@cs.umn.edu

A Spectrum of IV&V Modeling Techniques: Adapt the Suite of 6 Flight Guidance Models

Jimin Gao David Owen Mats Heimdahl

11 May 2003

Abstract

In preparation for a case study comparing (1) the University of Minnesota's NIMBUS toolset modeling technique and verification system (using the model checker NuSMV) to (2) West Virginia University's prototype modeling and analysis tool, Lurch, we have prepared a collection of flight guidance models to use in the experiments. To this effect, we have implemented an automatic translation procedure from NIMBUS input models to Lurch input models. We have used the automatic translation procedure to produce Lurch versions of the six flight guidance system models made available to the University of Minnesota by Rockwell Collins Inc. The translation scheme is presented in this report and the smallest flight guidance model is included as an example.

Contents

Chapter 1

Introduction

In preparation for a case study comparing (1) the University of Minnesota's NIMBUS toolset modeling technique and verification system (using the model checker NuSMV) to (2) West Virginia University's prototype modeling and analysis tool, Lurch, we have prepared a collection of flight guidance models to use in the experiments. To this effect, we have implemented an automatic translation procedure from NIMBUS input models to Lurch input models. We have used the automatic translation procedure to produce Lurch versions of the flight guidance system models made available to the University of Minnesota by Rockwell Collins Inc. There are six models of the same system, ranging from a small and abstract to very large and complex. WE have applied our translation framework to adapt all six flight guidance models to Lurch for experimentation. In this report we describe the RSML^{-e} modeling language used in NIMBUS, the Lurch input language and the automatic translation scheme. In addition, we provide the adaptation of the simplest flight guidance system as an example of how the translation is done.

Chapter 2

RSML^{-e} and NIMBUS

A detailed description of the RSML^{-e} modeling language and its simulation and translation environment NIMBUS is beyond the scope of this section. Below we present an overview of the language and tools—a detailed description is included in Appendix B for completeness of this report.

2.1 Framework

Figure 2.1 shows an overview of our verification framework. The user builds a behavioral model of the system in the fully formal and executable specification language RSML^{-e}. The specification is then fed to the NIMBUS simulator which checks that the specification is well formed and type correct. After the specification is checked, the user can translate the specification to the PVS or NuSMV input languages.

2.1.1 Overview of RSML^{-e}

RSML^{-e} stands for Requirements State Machine Language without Events. It is based on the Statecharts [3] like language Requirements State Machine Language (RSML) [4]. It is fully formal and a synchronous data-flow language without any internal broadcast events.

An RSML^{-e} specification consists of a collection of input variables, state variables, input/output interfaces, functions, macros, and constants; *input variables* are used to record the values observed in the environment, *state variables* are organized in a hierarchical fashion and are used to model various

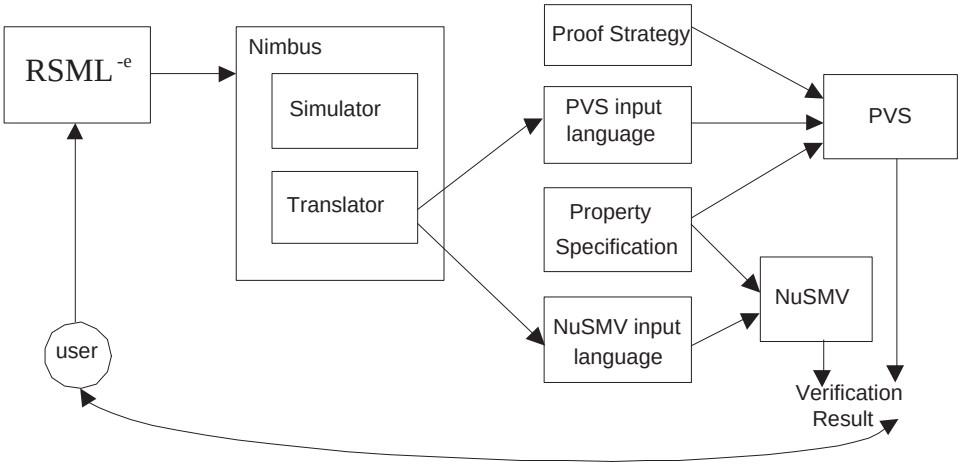


Figure 2.1: Verification Framework.

states of the control model, *interfaces* act as communication gateways to the external environment, and *functions and macros* encapsulate computations providing increased readability and ease of use.

Figure 2.2 shows a specification fragment of an RSML^{-e} specification of the Flight Guidance System[1]. The figure shows the definition of a state variable, `ROLL`. `ROLL` is the default lateral mode in the FGS mode logic. The state variable `ROLL` is declared as a child state of `Modes` and is active when the variable `Modes` has the value `On`—this notion of hierarchical variables provides the same abstractions and structuring mechanism as the AND and OR states in Statecharts , but the semantics is much simpler [6].

The conditions under which the state variable changes value are defined in the `TRANSITION` clauses in the definition. The condition tables are encoded in the macros, `Select_ROLL` and `Deselect_ROLL`. The use of macros not only improves the readability of the specifications but also helps localize errors and future changes. The tables are adopted from the original RSML notation– each column of truth values represents a conjunction of the propositions in the leftmost column (a '*' represents a "don't care" condition). If a table contains several columns, we take the disjunction of the columns; thus, the table is a way of expressing conditions in a disjunctive normal form. Sometimes we need to refer to values of the variables at a certain point in the variable

[1]We use here the ASCII version of RSML^{-e} since it is much more compact than the more readable typeset version.

```
STATE_VARIABLE ROLL : Base_State
        PARENT          : Modes.On
        INITIAL_VALUE   : UNDEFINED
        CLASSIFICATION  : State
        TRANSITION UNDEFINED TO Cleared IF NOT Select_ROLL()
        TRANSITION UNDEFINED TO Selected IF Select_ROLL()
        TRANSITION Cleared TO Selected IF Select_ROLL()
        TRANSITION Selected TO Cleared IF Deselect_ROLL()
END STATE_VARIABLE

MACRO Select_ROLL() :
        TABLE
                Is_No_Nonbasic_Lateral_Mode_Active()    : T;
                Modes = On                              : T;
        END TABLE
END MACRO

MACRO Deselect_ROLL() :
        TABLE
                When_Nonbasic_Lateral_Mode_Activated()  : T *;
                When(Modes = Off)                       : * T;
        END TABLE
END MACRO
```

Figure 2.2: A small portion of the FGS specification in RSML^{-e} .

history.

```
MACRO Were_Modes_Off() :
        PREV_STEP(Modes) = Off
END MACRO
```

In the above example, `PREV_STEP(Modes)` refers to the previous value of the state variable `Modes`.

Data-flow Semantics: RSML^{-e} transitions are purely condition-based and free of internal events—as soon as the guards in a variable definition can be evaluated, it will take on its new value. The variables are partially ordered based on the data dependency induced by the guard conditions—a similar semantics is adopted in the programming language Lustre [1]. Data-flow semantics removes complex issues caused by internal events, such as infinite triggering events or analysis of micro-steps [2], from the language.

Use of *Undefined* values: Startup behavior and behavior in the face of sensor failures pose particular challenges when specifying control systems—under these circumstances we simply do not know what the state of the

environment might be. RSML^{-e} supports modeling of this uncertainty by providing the concept of *Undefinedness*. One can explicitly specify the initial value of variables at startup to be *Undefined*, such as `ROLL=UNDEFINED` in Figure 2.2. Also, when a parent variable takes on a new value, each child variable of the parent value that was just changed are no longer relevant and must not be used—these child variables are *Undefined*. RSML^{-e} supports for both explicit and implicit *Undefinedness*.

Chapter 3

The Lurch Input Language

This section provides a short overview of the Lurch input language sufficient to explain the translation approach from RSML^{-e} to Lurch. The interested reader can find more information in [5].

3.1 State Machine Description

The Lurch input file is a list of finite-state machines. Different machines are separated by one or more blank lines. Each machine is a list of transitions, one per line. A transition takes the form:

```
<now>; <in>; <out>; <next>;
```

`<now>` is a single state in the machine being described, the current state. `<next>`, also, is a single state in the machine being described, the next state the machine goes to after the transition occurs. `<in>` and `<out>` are lists of inputs and outputs separated by commas. Each input and output is a state in some other machine (not the one being described). Basically, the transition says: if this machine is in state `<now>` and the conjunction of inputs listed in `<in>` is true, set the outputs listed in `<out>` to true and go to state `<next>`.

A minus sign '-' indicates that a field is blank. For example, if no input is required to trigger the transition (sometimes this is called a lambda or epsilon transition) it would be written as:

```
<now>; -; <out>; <next>;
```

A particular machine can only be in one state at a time. The translator needs to know which states belong to which machines so that they can be marked as mutually exclusive. The translator counts `<now>` and `<next>` as states in the machine being described, and marks them and all other states known to be in the same machine as mutually exclusive. If a state does not appear as `<now>` or `<next>` but is present in the machine, it can be declared (it is not necessary but doesn't hurt to declare all states this way):

```
<state>;
```

All declarations (including declarations implied by transitions) describing a machine must be in the same group, without any blank lines between them, in order to be recognized as a single machine.

3.2 Special State Marks For Record Keeping

Error states, or any states for which the user needs to see output as soon as they are reached by Lurch, are marked with "_" as the first character.

Progress states, significant because their presence inside a cycle means that cycle is not a "no-progress" cycle (a potential liveness violation) are marked with "+" as the first character.

3.3 C Function Call Extension

To make complex transition input and output more easy to model, and to make it possible to use existing code as part of the model, the input language has been extended. The new transition form is:

```
<now>; <in>,(<function call>); <out>,{<function call}; <next>;
```

The input file begins with arbitrary C code including any functions to be called by transitions. The C-code and finite-state machine sections are separated by the "%%" marker. Four special optional functions may be defined in the C-code section also:

- `void before(void)`: this function is called before each iteration of the search procedure. It should contain initializations for any C variables included in the model.

- `void after(void)`: this function is called after each iteration of the search procedure. It can be used, for example, to free memory allocated by C functions during the search iteration.

- `void lint(ce)`: for Lurch `prINT()` (no relation to the UNIX program). This function is called each time global state information is output to the counter example file. It can be used to track values of C variables in the counter example file.

- `void lash(int *)`: this function is called each time global state information is recorded by Lurch in an integer hash value. It is used to include C variables in the global state hash value, so that two global states, though they are identical in terms in finite-state machines, will get unique hash values if they have different C variable values.

3.4 Example

The example below shows an input model using many of the features described above.

```
/* Process Scheduling Algorithm from "The Model Checker SPIN"
(Holzmann) */

/* global C vars */
static int lk = 0, r_want = 0, r_lock = 0, sleep_q = 0;
static enum { Wakeme, Running } State = Running;

/* called at the beginning of each iteration */
void before()
{
  lk = r_want = r_lock = sleep_q = 0;
  State = Running;
}

/* put C var info into global state hash integer          */
/* void hash(char *, unsigned int *) is defined in node.c */
void lash(unsigned int *h)
{
```

```
  char c[16];

  sprintf(c, "%i", lk); hash(c, h);
  sprintf(c, "%i", r_want); hash(c, h);
  sprintf(c, "%i", r_lock); hash(c, h);
  sprintf(c, "%i", sleep_q); hash(c, h);
  sprintf(c, "%i", State); hash(c, h);
}

/* print C var info in counter example file */
void lint(FILE *ce)
{
  fprintf(ce, "   lk = %i\n", lk);
  fprintf(ce, "   r_want = %i\n", r_want);
  fprintf(ce, "   r_lock = %i\n", r_lock);
  fprintf(ce, "   sleep_q = %i\n", sleep_q);
  State == Wakeme ?
    fprintf(ce, "   State = Wakeme\n") :
    fprintf(ce, "   State = Running\n");
}
%%
(in this section any line without a semicolon is a comment)

+c14 is a progress state
_c14_assert_violated is an error state

c3;   (lk == 0);          {lk = 1;};            c11;
c5;   -;                  {r_want = 1;};        c6;
c6;   -;                  {State = Wakeme;};    c7;
c7;   -;                  {lk = 0;};            c8;
c8;   (State == Running); -;                    c3;
c11;  (r_lock == 1);      -;                    c5;
c11;  (r_lock == 0);      -;                    +c14;
+c14; (r_lock == 0);      -;                    c15;
+c14; (r_lock != 0);      _c14_assert_violated; +c14;
c15;  -;                  {r_lock = 1;};        c16;
c16;  -;                  {lk = 0;};            c3;
```

```
c14_assert_ok;
_c14_assert_violated;

s1;  -;                  {r_lock = 0;};      s2;
s2;  (lk == 0);          -;                  s15;
s6;  (sleep_q == 0);     {sleep_q = 1;};     s7;
s7;  -;                  {r_want = 0;};      s11;
s9;  -;                  {State = Running;}; s13;
s11; (State == Wakeme); -;                   s9;
s11; (State != Wakeme); -;                   s13;
s13; -;                  {sleep_q = 0;};     s1;
s15; (r_want == 1);      -;                  s6;
s15; (r_want != 1);      -;                  s1;
```

Chapter 4

Translation from RSML^{-e} to Lurch

This chapter describes the translation scheme from RSML^{-e} to the input language for Lurch random state search. Due the special purpose of the target language, it is often not expressive enough to represent a complete RSML^{-e} specification. Some non-critical information will be discarded during the translation process. We need to ensure that the generated code still represent equivalent state machines after this information loss. There are, however, also critical RSML^{-e} language features such as the integer variable type that cannot be supported by this translation at this stage, due to the common problem that numeric variables will cause for any finite state machine representation. For specifications that do use these features, we output an error message reporting that the translation cannot be completed.

4.1 Translation Scheme

4.1.1 Data Types

There are five data types in RSML^{-e} : Integer, Real, Time, Bool and enumerated types. This translation does not support the numeric types—only the Bool and enumerated types will be translated due to the limits of the Lurch input language. In RSML^{-e} , enumerated types are either explicitly defined or implied in a variable definition. In our target language, however, there is no concept of types, and enumeration symbols are simply used without

a type definition. Therefore the type definitions in RSML^{-e} can be ignored during the translation. Similarly, the Bool type can be viewed as a special type of enumeration, the values of which can be used in the Lurch input language without a type definition.

However, besides the state machine description, C language code can be embedded into the Lurch specification. To simplify the evaluation of state transition conditions, a C variable is declared for each state machine and keeps track of its most current state. Therefore, a C enumeration type declarations is needed for each RSML^{-e} type. For example, the RSML^{-e} type definition

```
TYPE_DEF DOIStatusType {On, Off}
```

will be translated into the following in the C code section:

```
enum DOIStatusType {DOIStatusType_On, DOIStatusType_Off};
```

4.1.2 Variables

Input Variables

Each input variable will be translated into an individual state machine in the target language. There are no state transitions for input variables in RSML^{-e} and they are assigned by the input interfaces. During the translation, random transitions will be specified for each input variable (the transition is nondeterministic, and can go from a state to any other state or stay in the old state). For example, the following input variable definition in RSML^{-e}

```
TYPE_DEF InhibitType { Inhibit, NoInhibit }

IN_VARIABLE InhibitSignal : InhibitType
   INITIAL_VALUE : NoInhibit
END IN_VARIABLE
```

will be translated into the following in the target language:

```
InhibitSignal=NoInhibit; InhibitSignal=NoInhibit; -;
    -, {InhibitSignal = InhibitType_Undefined;};
    InhibitSignal=Undefined;
InhibitSignal=NoInhibit; -;
```

```
    -, {InhibitSignal = InhibitType_Inhibit;};
    InhibitSignal=Inhibit;
InhibitSignal=NoInhibit; -;
    -, {InhibitSignal = InhibitType_NoInhibit;};
    InhibitSignal=NoInhibit;
InhibitSignal=Inhibit; -;
    -, {InhibitSignal = InhibitType_Undefined;};
    InhibitSignal=Undefined;
InhibitSignal=Inhibit; -;
    -, {InhibitSignal = InhibitType_Inhibit;};
    InhibitSignal=Inhibit;
InhibitSignal=Inhibit; -;
    -, {InhibitSignal = InhibitType_NoInhibit;};
    InhibitSignal=NoInhibit;
InhibitSignal=Undefined; -;
    -, {InhibitSignal = InhibitType_Undefined};
    InhibitSignal=Undefined;
InhibitSignal=Undefined; -;
    -, {InhibitSignal = InhibitType_Inhibit;};
    InhibitSignal=Inhibit;
InhibitSignal=Undefined; -;
    -, {InhibitSignal = InhibitType_NoInhibit;};
    InhibitSignal=NoInhibit;

InhibitSignal_prev=Undefined; InhibitSignal_prev=NoInhibit;
InhibitSignal=Inhibit;
    -, {InhibitSignal_prev = InhibitType_Inhibit;};
    InhibitSignal_prev=Inhibit;
InhibitSignal_prev=NoInhibit; InhibitSignal=Undefined;
    -, {InhibitSignal_prev = InhibitType_Undefined;};
    InhibitSignal_prev=Undefined;
InhibitSignal_prev=Inhibit; InhibitSignal=NoInhibit;
    -, {InhibitSingal_prev = InhibitType_NoInhibit;};
    InhibitSignal_prev=NoInhibit;
InhibitSignal_prev=Inhibit; InhibitSignal=Undefined;
    -, {InhibitSingal_prev = InhibitType_Undefined;};
    InhibitSignal_prev=Undefined;
InhibitSignal_prev=Undefined; InhibitSignal=Inhibit;
```

```
    -, {InhibitSingal_prev = InhibitType_Inhibit;};
    InhibitSignal_prev=Inhibit;
InhibitSignal_prev=Undefined; InhibitSignal=NoInhibit;
    -, {InhibitSingal_prev = InhibitType_NoInhibit;};
    InhibitSignal_prev=NoInhibit;
```

In this translation, `InhibitSignal_prev` is the state machine to keep track of the previous step value of `InhibitSignal`. It is needed if `InhibitSignal` is referenced by a SCR expression or by a *PREV_STEP* expression. Note that as part of the state transition output, a C language variable with the same name as the state machine is assigned to the value corresponding to the target state.

State Variables

Each state variable in RSML^{-e} will be translated into a state machine in the target language. For example, the state variable definition for `ASWOpModes` in the following RSML^{-e} example

```
STATE_VARIABLE ASWOpModes :
   VALUES : { OK, Inhibited, FailureDetected }
   PARENT : NONE
   INITIAL_VALUE : OK
   CLASSIFICATION : State

   EQUALS Inhibited IF
      TABLE
         InhibitSignal = Inhibit                : T;
         ivReset                                : F;
         DOI IN_STATE Failed                    : F;
         AltitudeStatus IN_STATE AltitudeBad    : F;
      END TABLE

   EQUALS FailureDetected IF
      TABLE
         DOI IN_STATE Failed                    : T *;
         AltitudeStatus IN_STATE AltitudeBad    : * T;
         ivReset                                : F F;
      END TABLE
```

```
   EQUALS OK IF
      TABLE
         InhibitSignal = Inhibit                : F *;
         ivReset                                : * T;
      END TABLE
END STATE_VARIABLE
```

can be translated into

```
ASWOpModes=OK; ASWOpModes=Undefined; -, (InhibitSignal == InhibitType_Inhibit &&
    !(ivReset == BOOLEAN_True) && DOI != DOI_type_Failed &&
    AltitudeStatus != AltitudeStatus_type_AltitudeBad);
    -, {ASWOpModes = ASWOpModes_type_Inhibited;};
    ASWOpModes=Inhibited;
ASWOpModes=OK; -, (InhibitSignal == InhibitType_Inhibit &&
    !(ivReset == BOOLEAN_True) && DOI != DOI_type_Failed &&
    AltitudeStatus != AltitudeStatus_type_AltitudeBad);
    -, {ASWOpModes = ASWOpModes_type_Inhibited;};
    ASWOpModes=Inhibited;
ASWOpModes=Inhibited; -, (InhibitSignal == InhibitType_Inhibit &&
    !(ivReset == BOOLEAN_True) && DOI != DOI_type_Failed &&
    AltitudeStatus != AltitudeStatus_type_AltitudeBad);
    -, {ASWOpModes = ASWOpModes_type_Inhibited;};
    ASWOpModes=Inhibited;
ASWOpModes=FailureDetected; -, (InhibitSignal ==
    InhibitType_Inhibit &&  !(ivReset == BOOLEAN_True) &&
    DOI != DOI_type_Failed &&
    AltitudeStatus != AltitudeStatus_type_AltitudeBad);
    -, {ASWOpModes = ASWOpModes_type_Inhibited;};
    ASWOpModes=Inhibited;
ASWOpModes=Undefined; -, ((DOI == DOI_type_Failed && !(ivReset ==
    BOOLEAN_True) || (AltitudeStatus ==
    AltutudeStatus_type_AltitudeBad && !(ivReset == BOOLEAN_True));
    -, {ASWOpModes = ASWOpModes_type_FailureDetected;};
    ASWOpModes=FailureDetected;
ASWOpModes=OK; -, ((DOI == DOI_type_Failed &&
    !(ivReset == BOOLEAN_True) ||
    (AltitudeStatus == AltutudeStatus_type_AltitudeBad &&
    !(ivReset == BOOLEAN_True));
    -, {ASWOpModes = ASWOpModes_type_FailureDetected;};
```

```
    ASWOpModes=FailureDetected;
ASWOpModes=Inhibiteded; -, ((DOI == DOI_type_Failed &&
    !(ivReset == BOOLEAN_True) || (AltitudeStatus ==
    AltutudeStatus_type_AltitudeBad &&
    !(ivReset == BOOLEAN_True));
    -, {ASWOpModes = ASWOpModes_type_FailureDetected;};
    ASWOpModes=FailureDetected;
ASWOpModes=FailureDetected; -, ((DOI == DOI_type_Failed &&
    !(ivReset == BOOLEAN_True) || (AltitudeStatus ==
    AltutudeStatus_type_AltitudeBad &&
    !(ivReset == BOOLEAN_True));
    -, {ASWOpModes = ASWOpModes_type_FailureDetected;};
    ASWOpModes=FailureDetected;
ASWOpModes=Undefined; -, ((!(InhibitSignal ==
    InhibitType_Inhibit)) || ivReset == BOOLEAN_True);
    -, {ASWOpModes = ASWOpModes_type_OK;};
    ASWOpModes=OK;
ASWOpModes=OK; -, ((!(InhibitSignal ==
    InhibitType_Inhibit)) || ivReset == BOOLEAN_True);
    -, {ASWOpModes = ASWOpModes_type_OK;};
    ASWOpModes=OK;
ASWOpModes=Inhibited; -, ((!(InhibitSignal ==
    InhibitType_Inhibit)) || ivReset == BOOLEAN_True);
    -, {ASWOpModes = ASWOpModes_type_OK;};
    ASWOpModes=OK;
ASWOpModes=FailureDetected; -, ((!(InhibitSignal ==
    InhibitType_Inhibit)) || ivReset == BOOLEAN_True);
    -, {ASWOpModes = ASWOpModes_type_OK;};
    ASWOpModes=OK;
```

In this translation, all the transition conditions are translated into embedded C code, and every transition is accompanied by a C variable assignment in the output action keeping track of the current state.

4.1.3 Expressions

Arithmetic and Relational Expressions

Since we do not support numeric types in this translation, the arithmetic and relational expressions, including $+$, $-$, $*$, $/$, $>$, $>=$, $<$, and $<=$ expressions, will not be supported either.

Variable Expressions

Because we cannot translate time to the target language, the variable time expressions ($TIME\_CHANGED$ and $TIME\_ASSIGNED$) will not be supported. In the variable value expressions, only expressions in the form $PREV\_VALUE(x, 0)$ will be supported, x being either a variable name or $PREV\_STEP$ expression of a variable. $PREV\_ASSIGN$ expressions will not be supported.

In order to support the $PREV\_STEP$ expression, aside from the state machines that represents the RSML^{-e} variables, we need to introduce a new state machine `x_prev` for a variable when needed.

Interface Expressions

Input interface expressions All the input interfaces in RSML^{-e} will be removed. The handlers within an interface, dealing with the assignment of input variables, will be translated into the state transitions of the input variables involved (See Section 4.1.5). The $LAST\_IO$ expression will not be supported.

Output interface expressions The output interfaces, responsible for the output of values of state variables, will be removed during the translation. The $LAST\_IO$ expression will not be supported.

AndOrTables

In RSML^{-e} , AndOrTables are used in the condition of a state transition or in the condition of an interface handler. The interface handler will also be translated into state transition statements in the target language. Since C predicates can be embedded in the state transition declaration of the Lurch input language, we choose to translate all AndOrTables into equivalent C boolean expressions (see example in Section 4.1.2).

Function and Macro Expressions

Our target language does not support the definition of functions and macros. Since functions may return numeric values they are not supported by this

translation. Macros will be translated into embedded C code function definitions in the NAYO input language, and a macro reference expression will be translated into a corresponding function call.

4.1.4 Constants

RSML^{-e} constants will be translated into NAYO state machines with only a initial state and no transitions.

4.1.5 Interfaces

Input Interfaces

All RSML^{-e} input interfaces will be removed during the translation. As the result, the input constraints (the handler conditions) will be lost during the translation. This is not a problem for the FGS models that we are going to experiment with initially, since all the handler conditions are `TRUE` in these models. However, eventually these conditions can be translated into state transition conditions by means of transforming all input variables into state variables and all message fields into input variables for the RSML^{-e} specification. This process will be implemented as an independent preprocessing pass in the NIMBUS framework that can be reused by different translators in the future.

Output Interfaces

All RSML^{-e} output interfaces will be removed during the translation. As the result, the output handler conditions will not be reflected in the translated NAYO specification. Similar to input interfaces, however, this information will not be lost if a preprocessing pass can transform all output message fields into state variables.

4.1.6 Messages

As previous stated, all messages and their associated fields will be removed during the translation unless a preprocessing pass is applied to transform them into input or state variables.

4.1.7 Functions and Macros

Functions in RSML^{-e} will not be supported by this translation. A macro definition will be translated into C language function definitions that can be imbedded into the NAYO specification. For example, the macro definition

```
MACRO AndAB() :
  TABLE:
    a : T;
    b : T;
  END TABLE
END MACRO
```

will be translated into

```
int AndAB() {
    return a && b;
}
```

4.2 Special Issues

4.2.1 Embedded C code

Besides the C predicates and C assignments imbedded in the state transition declarations of the translated Lurch specification, a special section before the state machine description is used for C variable declarations and initializations, and function definitions. In this section, a C variable with its associated enumeration type is declared for each state machine. A `before()` function initializes these C variables to the initial states before each search iteration. The translation for all RSML^{-e} macros is also located in this section.

4.2.2 Synchronization

Since Lurch uses an asynchronous search algorithm, while RSML^{-e} is a synchronous language where each variable is evaluated once per step in a dataflow order, in the translation we need to somehow force the the evaluation order of the Lurch state machines. This is achieved by adding an ordering state machine into the Lurch specification during the translation. For example, to translate a RSML^{-e} specification with variables `A`, `B`, and `C` (in evaluation order), we add state machine

```
0;
0; -; -; 1;
1; -; -; 2;
2; -; -; 0;
```

while in the translation for the variables, the order number is added to the transition conditions:

```
A=a1; 0, ...; -, ...; A=a2;
A=a2; 0, ...; -, ...; A=a3;
...

B=b1; 1, ...; -, ...; B=b2;
B=b2; 1, ...; -, ...; B=b3;
...

C=c1; 2, ...; -, ...; C=c2;
C=c2; 2, ...; -, ...; C=c3;
...
```

This way, all the state machines will be evaluated in dataflow order.

4.2.3 Property translation

All the temporal properties to be checked by Lurch will be manually translated into a state machine in the Lurch specification. The evaluation order number also needs to be added to the transition conditions, since many of the intermediate states during Lurch evaluation are not visible global states in RSML^{-e} . In the above example, only the global states when the ordering state machine is in state `0` are visible.

4.2.4 Boolean Undefinedness

As discussed in Appendix **??** RSML^{-e} uses a three valued logic for the Boolean type, `True`, `False` and `Undefined`. In the Lurch translation, the RSML^{-e} Boolean type is defined as the following:

```
enum BOOLEAN {BOOLEAN_False, BOOLEAN_True, BOOLEAN_Undefined};
```

The semantics of evaluating a Boolean variable with the value BOOLEAN_Undefined is in RSML^{-e} defined to be either True or False. Thus, the three values of a variable are mapped into the two values of Boolean expressions—RSML^{-e} simply treats an undefined value as an unknown Boolean value, either true or false. Thus, in the C translation, we cannot use the standard C Boolean operators since we are dealing with an enumerated type. Instead, we choose to treat the Booleans the same way they are dealt with in our translation to NuSMV—we explicitly check if a variable is equal to a truth value. For example, a RSML^{-e} Boolean expression `(A & B = Good)` containing a Boolean variable reference `A` would be translated into `(A == BOOLEAN_True && B == Good)` in the Lurch input language. Thus, if a Boolean variable has the value Undefined, it is *neither* true, nor false.

Bibliography

[1] S. Bensalem, P. Caspi, C. Parent-Vigouroux, and C. Dumas. A methodology for proving control systems with lustre and pvs. In *Proceedings of the Seventh Working Conference on Dependable Computing for Critical Applications (DCCA 7)*, 1999.

[2] W. Chan, R.J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J.D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.

[3] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[4] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese. Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering*, 20(9):684–706, September 1994.

[5] D. Owen and T. Menzies. Lurch: a lightweight alternative to model checking. In *Proceedign of the 2003 SOftware Engienerign and Knowledge Engineering Confernce*, 2003.

[6] Michael W. Whalen. A formal semantics for RSML^{-e}. Master's thesis, University of Minnesota, May 2000.

Appendix A

FGS01 Translation Example

To illustrate the translation from RSML^{-e} to Lurch, we include the full translation for one of the simplest FGS models—FGS-01.

A.1 FGS-01 in RSML^{-e}

```
/*************************************************************************/
/* Copyright  2001 Rockwell Collins, Inc. All rights reserved.       */
/*************************************************************************/

/*************************************************************************/
/* Toy FGS Requirements Specification Version 1                          */
/*                                                                       */
/* Version 0 consists of a simple Flight Director and the lateral        */
/* modes of Roll Hold (ROLL) and Heading Hold (HDG).                     */
/*                                                                       */
/* Version 1 adds the vertical modes of Pitch Hold (PTCH) and            */
/* Vertical Speed Hold (VS).                                             */
/*                                                                       */
/*************************************************************************/

/*************************************************************************/
/*L \section{Basic Definitions}                                        L*/
/*L This section defines types and constants                           L*/
/*L that are used throughout the specification. \bl                    L*/
/*************************************************************************/

        /*****************************************************************/
        /* The following types are the states of the hierarchical        */
        /* modes defined in the specification.                           */
        /*****************************************************************/
    TYPE_DEF On_Off         {Off, On}
    TYPE_DEF Base_State     {Cleared, Selected}
    TYPE_DEF Selected_State     {Armed, Active}

/*************************************************************************/
```

```
/*L \sectionp{Flight Director (FD)}
The Flight Director (FD) displays the pitch and roll guidance
commands to the pilot and copilot on the Primary Flight Display.
This component defines when the Flight Director guidance cues are
turned on and off.
L*/
/*********************************************************************/

    /*************************************************************/
    /*L \imports                                                 L*/
    /*************************************************************/
    MACRO When_Turn_FD_On() :
        TABLE
        When_FD_Switch_Pressed()        : T * *;
            When_Lateral_Mode_Manually_Selected()   : * T *;
        When_Vertical_Mode_Manually_Selected()  : * * T;
        END TABLE

            Purpose : &*L This event defines when the onside FD is
                to be turned on (i.e., displayed on the PFD). L*&

    END MACRO

    MACRO When_Turn_FD_Off(): When_FD_Switch_Pressed_Seen()

            Purpose : &*L This event defines when the onside FD is
                to be turned off (i.e., removed from the PFD). L*&
    END MACRO

    MACRO When_Lateral_Mode_Manually_Selected():
        When_HDG_Switch_Pressed_Seen()

            Purpose : &*L This event defines when a lateral
        mode is manually selected. L*&
    END MACRO

    MACRO When_Vertical_Mode_Manually_Selected():
        When_VS_Switch_Pressed_Seen()

            Purpose : &*L This event defines when a vertical
        mode is manually selected.  L*&

    END MACRO

    /*************************************************************/
    /*L \exports                                                 L*/
    /*************************************************************/
        STATE_VARIABLE Onside_FD: On_Off
        PARENT : None
        INITIAL_VALUE : Off
        CLASSIFICATION: State

        Transition Off TO On IF When_Turn_FD_On()

        Transition On TO Off IF When_Turn_FD_Off()

        Purpose : &*L This variable maintains the current
```

```
            state of the onside Flight Director. L*&

    END STATE_VARIABLE


/*************************************************************************/
/*L \sectionp{Flight Modes}
The flight modes determine which modes of
operation of the FGS are active and armed at any given moment.
These in term determine which flight control laws
are generating the commands directing the aircraft along the lateral
(roll) and vertical (pitch) axes.
This component encapsulates the
definitions of the lateral and vertical modes and defines how they
are synchronized.
L*/
/*************************************************************************/

    /*************************************************************/
    /*L \imports                                                 L*/
    /*************************************************************/
    MACRO When_Turn_Modes_On(): Onside_FD = On

            Purpose : &*L This event defines when the flight modes
                are to be turned on and displayed on the PFD. L*&
    END MACRO

    MACRO When_Turn_Modes_Off(): Onside_FD = Off

            Purpose : &*L This event defines when the flight
                modes are to be turned off and removed from the PFD. L*&
    END MACRO

    /*************************************************************/
    /*L \exports                                                 L*/
    /*************************************************************/
        STATE_VARIABLE Modes: On_Off
        PARENT : None
        INITIAL_VALUE : Off
        CLASSIFICATION: State

        TRANSITION Off TO On    IF When_Turn_Modes_On()

        TRANSITION On TO Off    IF When_Turn_Modes_Off()

            Purpose : &*L This variable maintains the current
                state of whether the mode annunciations are
                turned on or off. L*&
    END STATE_VARIABLE

    /*************************************************************/
    /* FD Cues On                                                */
    /*************************************************************/
    STATE_VARIABLE FD_Cues_On: Boolean
        PARENT : NONE
        INITIAL_VALUE : FALSE
        CLASSIFICATION: CONTROLLED
```

```
        EQUALS Onside_FD = On   IF TRUE

        Purpose : &*L Indicates if the FD Guidance cues
        should be displayed on the PFD. L*&

    END STATE_VARIABLE

    /*************************************************************/
    /* Modes On Off                                              */
    /*************************************************************/
    STATE_VARIABLE Mode_Annunciations_On: Boolean
        PARENT : NONE
        INITIAL_VALUE : FALSE
        CLASSIFICATION: CONTROLLED

        EQUALS Modes = On   IF TRUE

        Purpose : &*L Indicates if the mode annunications
        should be displayed on the PFD. L*&

    END STATE_VARIABLE

/*********************************************************************/
/*L \subsectionp{Lateral Modes}
The lateral modes select the control laws generating commands
directing the aircraft along the lateral, or roll, axis.
This component encapsulates the specific lateral modes
present in this aircraft and defines how they are synchronized.
L*/
/*********************************************************************/

    /*************************************************************/
    /*L \encapsulated                                          L*/
    /*************************************************************/
    MACRO When_Nonbasic_Lateral_Mode_Activated() : When_HDG_Activated()

        Purpose : &*L This event ocurrs when a new lateral
                mode other than the basic mode becomes active. It is
        used to deselect active or armed modes. L*&

        Comment: &*L Basic mode is excluded to avoid a
        cyclic dependency in the definition of this macro. L*&
    END MACRO

    MACRO Is_No_Nonbasic_Lateral_Mode_Active() : NOT Is_HDG_Active

        Purpose : &*L This condition indicates if no lateral
        mode except basic mode is active. It is used to
        trigger the activation of the basic lateral mode. L*&

        Comment: &*L Basic mode is excluded to avoid a
        cyclic dependency in the definition of this macro. L*&
    END MACRO

/*********************************************************************/
/*L \subsubsectionp{Roll Hold (ROLL) Mode}
```

```
In Roll Hold mode the FGS generates guidance commands to hold the
aircraft at a fixed bank angle.
Roll Hold mode is the basic lateral mode and is always active when
the modes are displayed and no other lateral mode is active.
L*/
/*********************************************************************/

    /************************************************************/
    /*L \imports                                                L*/
    /************************************************************/
    MACRO Select_ROLL() :
        TABLE
            Is_No_Nonbasic_Lateral_Mode_Active()    : T;
            Modes = On              : T;
        END TABLE

        Purpose : &*L This event defines when Roll Hold mode
        is to be selected. Roll Hold mode is the basic, or default,
        mode and is selected whenever the mode annunciations
        are on and no other lateral mode is active. L*&

        Comment : &*L To avoid cyclic dependencies, the
        only way to select Roll Hold mode is to deselect
        the active lateral mode, which will automatically
        activate Roll Hold. L*&
    END MACRO

    MACRO Deselect_ROLL() :
        TABLE
            When_Nonbasic_Lateral_Mode_Activated()  : T *;
            When(Modes = Off)           : * T;
        END TABLE

        Purpose : &*L The event defines when Roll Hold mode is
        to be deselected. This occurs when a new lateral mode is
        activated or the modes are turned off. L*&
    END MACRO

    /************************************************************/
    /*L \exports                                                L*/
    /************************************************************/
    STATE_VARIABLE Is_ROLL_Selected: Boolean
        PARENT : NONE
        INITIAL_VALUE : FALSE
        CLASSIFICATION: CONTROLLED

        EQUALS ..ROLL = Selected IF TRUE

        Purpose : &*L Indicates if ROLL mode is selected. L*&

    END STATE_VARIABLE

    STATE_VARIABLE Is_ROLL_Active: Boolean
        PARENT : NONE
        INITIAL_VALUE : FALSE
        CLASSIFICATION: CONTROLLED
```

```
        EQUALS ..ROLL = Selected IF TRUE

        Purpose : &*L Indicates if ROLL mode is active. L*&

        Comment : &*L Even though ROLL Selected and ROLL Active are
        the same thing, this variable is introduced to maintain a
        common interface across modes. L*&

    END STATE_VARIABLE

    /*************************************************************/
    /*L \encapsulated                                            L*/
    /*************************************************************/
    STATE_VARIABLE ROLL : Base_State
        PARENT : Modes.On
        INITIAL_VALUE   : UNDEFINED
        CLASSIFICATION  : State

        TRANSITION UNDEFINED TO Cleared IF NOT Select_ROLL()

        TRANSITION UNDEFINED TO Selected IF Select_ROLL()

        TRANSITION Cleared TO Selected IF Select_ROLL()

        TRANSITION Selected TO Cleared IF Deselect_ROLL()

        Purpose : &*L This variable maintains the current base
        state of Roll Hold mode, i.e., whether it is
        cleared or selected. L*&

    END STATE_VARIABLE

/*************************************************************************/
/*L \subsubsectionp{Heading Select (HDG) Mode}
In Heading Select mode, the FGS provides guidance commands to
to track the Selected Heading displayed on the PFD.
L*/
/*************************************************************************/

    /*************************************************************/
    /*L \imports                                                 L*/
    /*************************************************************/
    MACRO Select_HDG() : When_HDG_Switch_Pressed_Seen()

        Purpose : &*L This event defines when Heading Select
        mode is to be selected. L*&
    END MACRO

    MACRO Deselect_HDG() :
        TABLE
            When_HDG_Switch_Pressed_Seen()      : T * *;
            When_Nonbasic_Lateral_Mode_Activated()  : * T *;
            When(Modes = Off)           : * * T;
        END TABLE

        Purpose : &*L This event defines when Heading Select mode
        is to be deselected.  L*&
```

```
END MACRO

/*************************************************************/
/*L \exports                                                 L*/
/*************************************************************/
STATE_VARIABLE Is_HDG_Selected: Boolean
    PARENT : NONE
    INITIAL_VALUE : FALSE
    CLASSIFICATION: CONTROLLED

    EQUALS ..HDG = Selected IF TRUE

    Purpose : &*L Indicates if Hdg Mode is selected. L*&

END STATE_VARIABLE

STATE_VARIABLE Is_HDG_Active: Boolean
    PARENT : NONE
    INITIAL_VALUE : FALSE
    CLASSIFICATION: CONTROLLED

    EQUALS ..HDG = Selected IF TRUE

    Purpose : &*L Indicates if HDG Mode is active. L*&

    Comment : &*L Even though HDG Selected and HDG Active are
    the same thing, this variable is introduced to maintain a
    common interface across modes. L*&

END STATE_VARIABLE

MACRO When_HDG_Activated() :
    TABLE
        Select_HDG()                : T;
        PREV_STEP(..HDG) = Selected : F;
    END TABLE

    Purpose : &*L This signal occurs when Heading Select mode
            is activated.  L*&

    Comment : &*L This event is defined this way to avoid
    circular dependencies. It would be preferable to define
    it as When(HDG = Selected). L*&
END MACRO

/*************************************************************/
/*L \encapsulated                                            L*/
/*************************************************************/
STATE_VARIABLE HDG : Base_State
    PARENT : Modes.On
    INITIAL_VALUE :  UNDEFINED
    CLASSIFICATION : State

    Purpose : &*L This variable maintains the current base
    state of Heading Select mode, i.e., whether it is
    cleared or selected. L*&
```

```
        TRANSITION UNDEFINED TO Cleared IF NOT Select_HDG()

        TRANSITION UNDEFINED TO Selected IF Select_HDG()

        TRANSITION Cleared TO Selected IF Select_HDG()

        TRANSITION Selected TO Cleared IF Deselect_HDG()

    END STATE_VARIABLE

/*********************************************************************/
/*L \subsectionp{Vertical Modes}
The vertical modes select the control laws generating commands
directing the aircraft along the vertical, or pitch, axis.
This component encapsulates the specific vertical modes
present in this aircraft and defines how they are synchronized.
L*/
/*********************************************************************/

    /*************************************************************/
    /*L \encapsulated                                            L*/
    /*************************************************************/
    MACRO When_Nonbasic_Vertical_Mode_Activated() : When_VS_Activated()

        Purpose : &*L This event indicates when a new vertical
                mode other than the basic mode becomes active. It is
        used to deselect active or armed modes. L*&

        Comment: &*L Basic mode is excluded to avoid a
        cyclic dependency in the definition of this macro. L*&

    END MACRO

    MACRO Is_No_Nonbasic_Vertical_Mode_Active() : NOT Is_VS_Active

        Purpose : &*L This condition indicates if no vertical
        mode except basic mode is active. It is used to
        trigger the activation of the basic lateral mode. L*&

        Comment: &*L Basic mode is excluded to avoid a
        cyclic dependency in the definition of this macro. L*&

    END MACRO

/*********************************************************************/
/*L \subsubsectionp{Pitch Hold (PITCH) Mode}
In Pitch Hold mode the FGS generates guidance commands to hold the
aircraft at a fixed pitch angle.
Pitch Hold mode is the basic vertical mode and is always active when
the modes are displayed and no other vertical mode is active.
L*/
/*********************************************************************/

    /*************************************************************/
    /*L \imports                                                 L*/
    /*************************************************************/
    MACRO Select_PITCH() :
```

```
    TABLE
        Is_No_Nonbasic_Vertical_Mode_Active()   : T;
        Modes = On              : T;
    END TABLE

    Purpose : &*L Pitch Hold mode is the basic, or default,
    mode and is selected whenever the mode annunciations
    are on and no other vertical mode is active. L*&

    Comment : &*L To avoid cyclic dependencies, the
    only way to select Pitch Hold mode is to deselect
    the active vertical mode, which will automatically
    activate Pitch Hold mode. L*&

END MACRO

MACRO Deselect_PITCH() :
    TABLE
        When_Nonbasic_Vertical_Mode_Activated() : T *;
        When(Modes = Off)           : * T;
    END TABLE

    Purpose : &*L Pitch Hold mode is deselected when:
    a new vertical mode is activated or
    the modes are turned off. L*&

END MACRO

/*************************************************************/
/*L \exports                                                 L*/
/*************************************************************/
STATE_VARIABLE Is_PITCH_Selected: Boolean
    PARENT : NONE
    INITIAL_VALUE : FALSE
    CLASSIFICATION: CONTROLLED

    EQUALS ..PITCH = Selected IF TRUE

    Purpose : &*L Indicates if PITCH mode is selected. L*&

END STATE_VARIABLE

STATE_VARIABLE Is_PITCH_Active: Boolean
    PARENT : NONE
    INITIAL_VALUE : FALSE
    CLASSIFICATION: CONTROLLED

    EQUALS ..PITCH = Selected IF TRUE

    Purpose : &*L Indicates if PITCH mode is active. L*&

    Comment : &*L Even though PITCH Selected and PITCH Active are
    the same thing, this variable is introduced to maintain a
    common interface across modes. L*&

END STATE_VARIABLE
```

```
    /*************************************************************/
    /*L \encapsulated                                            L*/
    /*************************************************************/
    STATE_VARIABLE PITCH : Base_State
        PARENT : Modes.On
        INITIAL_VALUE   : UNDEFINED
        CLASSIFICATION  : State

        TRANSITION UNDEFINED TO Cleared IF NOT Select_PITCH()

        TRANSITION UNDEFINED TO Selected IF Select_PITCH()

        TRANSITION Cleared TO Selected IF Select_PITCH()

        TRANSITION Selected TO Cleared IF Deselect_PITCH()

        Purpose : &*L This variable maintains the current base
        state of Pitch Hold mode, i.e., whether it is
        cleared or selected. L*&

    END STATE_VARIABLE


/*********************************************************************/
/*L \subsubsectionp{Vertical Speed (VS) Mode}
In Vertical Speed mode, the FGS provides pitch guidance commands to
to hold the aircraft to the Vertical Speed (VS) reference.
L*/
/*********************************************************************/

    /*************************************************************/
    /*L \imports                                                 L*/
    /*************************************************************/
    MACRO Select_VS() : When_VS_Switch_Pressed_Seen()

        Purpose : &*L This event defines when Vertical Speed
        mode is to be selected. L*&
    END MACRO

    MACRO Deselect_VS() :
        TABLE
        When_VS_Switch_Pressed_Seen()       : T * *;
        When_Nonbasic_Vertical_Mode_Activated() : * T *;
        When(Modes = Off)           : * * T;
        END TABLE

        Purpose : &*L This event defines when Vertical Speed mode
        is to be deselected.  L*&

    END MACRO

    /*************************************************************/
    /*L \exports                                                 L*/
    /*************************************************************/
    STATE_VARIABLE Is_VS_Selected: Boolean
        PARENT : NONE
        INITIAL_VALUE : FALSE
```

```
    CLASSIFICATION: CONTROLLED

    EQUALS ..VS = Selected IF TRUE

    Purpose : &*L Indicates if VS mode is selected. L*&

END STATE_VARIABLE

STATE_VARIABLE Is_VS_Active: Boolean
    PARENT : NONE
    INITIAL_VALUE : FALSE
    CLASSIFICATION: CONTROLLED

    EQUALS ..VS = Selected IF TRUE

    Purpose : &*L Indicates if VS mode is active. L*&

    Comment : &*L Even though VS Selected and VS Active are
    the same thing, this variable is introduced to maintain a
    common interface across modes. L*&

END STATE_VARIABLE

MACRO When_VS_Activated() :
    TABLE
        Select_VS()             : T;
        PREV_STEP(..VS) = Selected  : F;
    END TABLE
    Purpose : &*L This signal occurs when Vertical Speed mode
            is activated.  L*&

    Comment : &*L This event is defined this way to avoid
    circular dependencies. It would be preferable to define
    it as When(VS = Selected). L*&
END MACRO

/*************************************************************/
/*L \encapsulated                                          L*/
/*************************************************************/
STATE_VARIABLE VS: Base_State
    PARENT : Modes.On
    INITIAL_VALUE :  UNDEFINED
    CLASSIFICATION : State

    TRANSITION UNDEFINED TO Cleared IF NOT Select_VS()

    TRANSITION UNDEFINED TO Selected IF Select_VS()

    TRANSITION Cleared TO Selected IF Select_VS()

    TRANSITION Selected to Cleared IF Deselect_VS()

    Purpose : &*L This variable maintains the current base
    state of Vertical Speed mode, i.e., whether it is
    cleared or selected. L*&

END STATE_VARIABLE
```

```
/*********************************************************************/
/*L \sectionp{Flight Control Panel (FCP)}
L*/
/*********************************************************************/

    /*************************************************************/
    /*L \exports                                                 L*/
    /*************************************************************/
    MACRO When_FD_Switch_Pressed() : When(FD_Switch = ON)

    Purpose : &*L This event indicates when the FD switch
    associated with this FGS is pressed. L*&

    Comment: &*L This is redefined as a macro to simplify verification. L*&

    END MACRO

    MACRO When_FD_Switch_Pressed_Seen():
        TABLE
            When_FD_Switch_Pressed()                    : T;
            No_Higher_Event_Than_FD_Switch_Pressed()    : T;
        END TABLE

    Purpose : &*L This event indicates when the FD switch is pressed
    and no higher priority event has occurred. L*&

    END MACRO

    MACRO No_Higher_Event_Than_FD_Switch_Pressed():
        TABLE
            When_HDG_Switch_Pressed()           : F;
            No_Higher_Event_Than_HDG_Switch_Pressed()   : T;
            When_VS_Switch_Pressed()            : F;
            No_Higher_Event_Than_VS_Switch_Pressed()    : T;
        END TABLE

    Purpose : &*L This event occurs when no event with a priority
    higher than pressing the FD switch has occurred. L*&

    END MACRO

    MACRO When_HDG_Switch_Pressed() : When(HDG_Switch = ON)

    Purpose : &*L This event indicates when the HDG switch is pressed. L*&

    Comment: &*L This is redefined as a macro to simplify verification. L*&

    END MACRO

    MACRO When_HDG_Switch_Pressed_Seen()    :
        TABLE
            When_HDG_Switch_Pressed()           : T;
            No_Higher_Event_Than_HDG_Switch_Pressed()   : T;
        END TABLE

    Purpose : &*L This event indicates when the HDG switch
```

```
pressed and no higher priority event has occurred. L*&

END MACRO

MACRO No_Higher_Event_Than_HDG_Switch_Pressed(): TRUE

Purpose : &*L This event occurs when no event with a priority
higher than pressing the HDG switch has occurred. L*&

END MACRO

MACRO When_VS_Switch_Pressed() : When(VS_Switch = ON)

Purpose : &*L This event indicates when the VS switch is pressed. L*&

Comment: &*L This is redefined as a macro to simplify verification. L*&

END MACRO

MACRO When_VS_Switch_Pressed_Seen() :
    TABLE
        When_VS_Switch_Pressed()            : T;
        No_Higher_Event_Than_VS_Switch_Pressed()    : T;
    END TABLE

Purpose : &*L This event indicates when the VS switch
pressed and no higher priority event has occurred. L*&

END MACRO

MACRO No_Higher_Event_Than_VS_Switch_Pressed(): TRUE

Purpose : &*L This event occurs when no event with a priority
higher than pressing the VS switch has occurred. L*&

END MACRO

/*************************************************************/
/*L \encapsulated                                        L*/
/*************************************************************/
TYPE_DEF Switch {OFF, ON}
TYPE_DEF Lamp {OFF, ON}

/*************************************************************/
/* FD Switch                                                 */
/*************************************************************/
IN_VARIABLE FD_Switch: Switch
    INITIAL_VALUE : UNDEFINED
    CLASSIFICATION: MONITORED
    Purpose : &*L Holds the last sensed position of the
    FD switch associated with this FGS. L*&

END IN_VARIABLE

/*************************************************************/
/* HDG Switch                                                */
/*************************************************************/
```

```
    IN_VARIABLE HDG_Switch: Switch
        INITIAL_VALUE : UNDEFINED
        CLASSIFICATION: MONITORED

        Purpose : &*L Holds the last sensed position of the
        HDG switch. L*&
    END IN_VARIABLE

    /*************************************************************/
    /* HDG Lamp                                                  */
    /*************************************************************/
    STATE_VARIABLE HDG_Lamp: Lamp
        PARENT : NONE
        INITIAL_VALUE : OFF
        CLASSIFICATION: CONTROLLED

        EQUALS ON   IF ..HDG = Selected
        EQUALS OFF  IF NOT (..HDG = Selected)

        Purpose : &*L Indicates if the HDG switch lamp
        on the FCP should be on or off. L*&

    END STATE_VARIABLE

    /*************************************************************/
    /* VS Switch                                                 */
    /*************************************************************/
    IN_VARIABLE VS_Switch: Switch
        INITIAL_VALUE : UNDEFINED
        CLASSIFICATION: MONITORED

        Purpose : &*L Holds the last sensed position of the
        VS switch. L*&
    END IN_VARIABLE

    /*************************************************************/
    /* VS Lamp                                                   */
    /*************************************************************/
    STATE_VARIABLE VS_Lamp: Lamp
        PARENT : NONE
        INITIAL_VALUE : OFF
        CLASSIFICATION: CONTROLLED

        EQUALS ON   IF ..VS = Selected
        EQUALS OFF  IF NOT (..VS = Selected)

        Purpose : &*L Indicates if the VS switch lamp
        should be on or off. L*&

    END STATE_VARIABLE

/*********************************************************************/
/*L \sectionp{FGS Inputs}
This section defines the physical interface for all inputs to the FGS.
The input variables associated with these fields are defined in the
part of the specification to which they are logically related.
L*/
```

```
/*********************************************************************/

    /**** Autocoded inputs for [ToyFGS01] interface [This] ****/
    MESSAGE This_Input_Msg {
            FdSwi IS Switch,
            HdgSwi IS Switch,
            VsSwi IS Switch}

    /**** Autocoded inputs for [ToyFGS01] interface [This] ****/
    IN_INTERFACE This_Input :
        MIN_SEP : UNDEFINED
        MAX_SEP : UNDEFINED

        INPUT_ACTION : READ(This_Input_Msg)
        HANDLER:
                CONDITION : TRUE
                    ASSIGNMENT
                        FD_Switch   := FdSwi,
                        HDG_Switch  := HdgSwi,
                        VS_Switch   := VsSwi
                    END ASSIGNMENT
                END HANDLER
        END IN_INTERFACE


/*********************************************************************/
/*L \sectionp{FGS Outputs}
This section defines the physical interface for all outputs from the
FGS. The output variables associated with these fields are defined in
the part of the specification to which they are logically related.
L*/
/*********************************************************************/

    /**** Autocoded outputs for [ToyFGS01] interface [This] ****/
    MESSAGE This_Output_Msg {
            FdOn IS Boolean,
            FGSActive IS Boolean,
            HdgLamp IS Lamp,
            HdgSel IS Boolean,
            ModesOn IS Boolean,
            PthSel IS Boolean,
            RollSel IS Boolean,
            VsLamp IS Lamp,
            VsSel IS Boolean}

    /**** Autocoded outputs for [ToyFGS01] interface [This] ****/
    OUT_INTERFACE This_Output:
        MIN_SEP : UNDEFINED
        MAX_SEP : UNDEFINED

        OUTPUT_ACTION : PUBLISH(This_Output_Msg)
            HANDLER:
                    CONDITION : TABLE
                        CHANGED(FD_Cues_On)     : T * * * * * * *;
                        CHANGED(HDG_Lamp)       : * T * * * * * *;
                        CHANGED(Is_HDG_Selected)        : * * T * * * * *;
                            CHANGED(Mode_Annunciations_On)  : * * * T * * * *;
```

```
                    CHANGED(Is_PITCH_Selected)      : * * * * T * * *;
                    CHANGED(Is_ROLL_Selected)       : * * * * * T * *;
                    CHANGED(VS_Lamp)        : * * * * * * T *;
                    CHANGED(Is_VS_Selected)     : * * * * * * * T;
                END TABLE
            ASSIGNMENT
                FdOn        := FD_Cues_On,
                FGSActive   := TRUE,
                HdgLamp     := HDG_Lamp,
                HdgSel      := Is_HDG_Selected,
                ModesOn     := Mode_Annunciations_On,
                PthSel      := Is_PITCH_Selected,
                RollSel     := Is_ROLL_Selected,
                VsLamp      := VS_Lamp,
                VsSel       := Is_VS_Selected
            END ASSIGNMENT
            ACTION : SEND
        END HANDLER
    END OUT_INTERFACE
```

A.2 FGS-01 in Lurch

The RSML^{-e} model in the section above is translated into the following Lurch input specification.

```
enum Switch {Switch_OFF, Switch_ON, Switch_Undefined} FD_Switch, FD_Switch_prev, HDG_Switch,
    HDG_Switch_prev, VS_Switch, VS_Switch_prev;
enum Base_State {Base_State_Cleared, Base_State_Selected, Base_State_Undefined} HDG, HDG_prev,
    ROLL, ROLL_prev, VS, VS_prev, PITCH, PITCH_prev;
enum On_Off {On_Off_Off, On_Off_On, On_Off_Undefined} Onside_FD, Onside_FD_prev, Modes,
    Modes_prev;
enum BOOLEAN {BOOLEAN_False = 0, BOOLEAN_True = 1, BOOLEAN_Undefined = 2} FD_Cues_On,
    FD_Cues_On_prev, Mode_Annunciations_On, Mode_Annunciations_On_prev, Is_ROLL_Selected,
    Is_ROLL_Selected_prev, Is_ROLL_Active, Is_ROLL_Active_prev, Is_HDG_Selected,
    Is_HDG_Selected_prev, Is_VS_Selected, Is_VS_Selected_prev, Is_VS_Active, Is_VS_Active_prev,
    Is_HDG_Active, Is_HDG_Active_prev, Is_PITCH_Selected, Is_PITCH_Selected_prev,
    Is_PITCH_Active, Is_PITCH_Active_prev;
enum Lamp {Lamp_OFF, Lamp_ON, Lamp_Undefined} HDG_Lamp, HDG_Lamp_prev, VS_Lamp, VS_Lamp_prev;

void before(void) {
    FD_Switch = Switch_Undefined;
    FD_Switch_prev = Switch_Undefined;
    HDG_Switch = Switch_Undefined;
    HDG_Switch_prev = Switch_Undefined;
    VS_Switch = Switch_Undefined;
    VS_Switch_prev = Switch_Undefined;
    HDG = Base_State_Undefined;
    HDG_prev = Base_State_Undefined;
    ROLL = Base_State_Undefined;
    ROLL_prev = Base_State_Undefined;
    VS = Base_State_Undefined;
    VS_prev = Base_State_Undefined;
    PITCH = Base_State_Undefined;
```

```
    PITCH_prev = Base_State_Undefined;
    Onside_FD = On_Off_Off;
    Onside_FD_prev = On_Off_Undefined;
    Modes = On_Off_Off;
    Modes_prev = On_Off_Undefined;
    FD_Cues_On = BOOLEAN_False;
    FD_Cues_On_prev = BOOLEAN_Undefined;
    Mode_Annunciations_On = BOOLEAN_False;
    Mode_Annunciations_On_prev = BOOLEAN_Undefined;
    Is_ROLL_Selected = BOOLEAN_False;
    Is_ROLL_Selected_prev = BOOLEAN_Undefined;
    Is_ROLL_Active = BOOLEAN_False;
    Is_ROLL_Active_prev = BOOLEAN_Undefined;
    Is_HDG_Selected = BOOLEAN_False;
    Is_HDG_Selected_prev = BOOLEAN_Undefined;
    Is_VS_Selected = BOOLEAN_False;
    Is_VS_Selected_prev = BOOLEAN_Undefined;
    Is_VS_Active = BOOLEAN_False;
    Is_VS_Active_prev = BOOLEAN_Undefined;
    Is_HDG_Active = BOOLEAN_False;
    Is_HDG_Active_prev = BOOLEAN_Undefined;
    Is_PITCH_Selected = BOOLEAN_False;
    Is_PITCH_Selected_prev = BOOLEAN_Undefined;
    Is_PITCH_Active = BOOLEAN_False;
    Is_PITCH_Active_prev = BOOLEAN_Undefined;
    HDG_Lamp = Lamp_OFF;
    HDG_Lamp_prev = Lamp_Undefined;
    VS_Lamp = Lamp_OFF;
    VS_Lamp_prev = Lamp_Undefined;
}

%%
Onside_FD=Off;
Onside_FD=Off; 0, ((!(FD_Switch_prev == Switch_ON) && (FD_Switch == Switch_ON)) ||
    (!(HDG_Switch_prev == Switch_ON) && (HDG_Switch == Switch_ON)) ||
    (!(VS_Switch_prev == Switch_ON) && (VS_Switch == Switch_ON)));
    -, {Onside_FD = On_Off_On;}; Onside_FD=On;
Onside_FD=On; 0, ((!(FD_Switch_prev == Switch_ON) && (FD_Switch == Switch_ON)) &&
    (!(!(HDG_Switch_prev == Switch_ON) && (HDG_Switch == Switch_ON))) &&
    (!(!(VS_Switch_prev == Switch_ON) && (VS_Switch == Switch_ON))));
    -, {Onside_FD = On_Off_Off;}; Onside_FD=Off;


Onside_FD_prev=Undefined;
Onside_FD_prev=On; Onside_FD=Off, 19; -, {Onside_FD_prev = On_Off_Off;}; Onside_FD_prev=Off;
Onside_FD_prev=Undefined; Onside_FD=Off, 19; -, {Onside_FD_prev = On_Off_Off;};
    Onside_FD_prev=Off;
Onside_FD_prev=Off; Onside_FD=Undefined, 19; -, {Onside_FD_prev= On_Off_Undefined;};
    Onside_FD_prev=Undefined;
Onside_FD_prev=Off; Onside_FD=On, 19; -, {Onside_FD_prev = On_Off_On;}; Onside_FD_prev=On;
Onside_FD_prev=Undefined; Onside_FD=On, 19; -, {Onside_FD_prev = On_Off_On;}; Onside_FD_prev=On;
Onside_FD_prev=On; Onside_FD=Undefined, 19; -, {Onside_FD_prev= On_Off_Undefined;};
    Onside_FD_prev=Undefined;

Modes=Off;
Modes=Off; 1, ((Onside_FD == On_Off_On)); -, {Modes = On_Off_On;}; Modes=On;
Modes=On; 1, ((Onside_FD == On_Off_Off)); -, {Modes = On_Off_Off;}; Modes=Off;
```

```
Modes_prev=Undefined;
Modes_prev=On; Modes=Off, 19; -, {Modes_prev = On_Off_Off;}; Modes_prev=Off;
Modes_prev=Undefined; Modes=Off, 19; -, {Modes_prev = On_Off_Off;}; Modes_prev=Off;
Modes_prev=Off; Modes=Undefined, 19; -, {Modes_prev= On_Off_Undefined;}; Modes_prev=Undefined;
Modes_prev=Off; Modes=On, 19; -, {Modes_prev = On_Off_On;}; Modes_prev=On;
Modes_prev=Undefined; Modes=On, 19; -, {Modes_prev = On_Off_On;}; Modes_prev=On;
Modes_prev=On; Modes=Undefined, 19; -, {Modes_prev= On_Off_Undefined;}; Modes_prev=Undefined;

HDG=Undefined;
HDG=Cleared; 2, ((Modes == On_Off_Undefined) || (Modes != On_Off_On));
    -, {HDG = Base_State_Undefined;}; HDG=Undefined;
HDG=Selected; 2, ((Modes == On_Off_Undefined) || (Modes != On_Off_On));
    -, {HDG = Base_State_Undefined;}; HDG=Undefined;
HDG=Undefined; 2, ((!(!(HDG_Switch_prev == Switch_ON) && (HDG_Switch == Switch_ON))) &&
    (Modes != On_Off_Undefined) && (Modes == On_Off_On)); -, {HDG = Base_State_Cleared;};
    HDG=Cleared;
HDG=Undefined; 2, ((!(HDG_Switch_prev == Switch_ON) && (HDG_Switch == Switch_ON)) &&
    (Modes != On_Off_Undefined) && (Modes == On_Off_On)); -, {HDG = Base_State_Selected;};
    HDG=Selected;
HDG=Cleared; 2, ((!(HDG_Switch_prev == Switch_ON) && (HDG_Switch == Switch_ON)) &&
    (Modes != On_Off_Undefined) && (Modes == On_Off_On)); -, {HDG = Base_State_Selected;};
    HDG=Selected;
HDG=Selected; 2, ((!(HDG_Switch_prev == Switch_ON) && (HDG_Switch == Switch_ON)) &&
    (Modes != On_Off_Undefined) && (Modes == On_Off_On) || (!(HDG_Switch_prev == Switch_ON) &&
    (HDG_Switch == Switch_ON)) && (!(HDG_prev == Base_State_Selected)) &&
    (Modes != On_Off_Undefined) && (Modes == On_Off_On) || (!(Modes_prev == On_Off_Off) &&
    (Modes == On_Off_Off)) && (Modes != On_Off_Undefined) && (Modes == On_Off_On));
    -, {HDG = Base_State_Cleared;}; HDG=Cleared;


HDG_prev=Undefined;
HDG_prev=Selected; HDG=Cleared, 19; -, {HDG_prev = Base_State_Cleared;}; HDG_prev=Cleared;
HDG_prev=Undefined; HDG=Cleared, 19; -, {HDG_prev = Base_State_Cleared;}; HDG_prev=Cleared;
HDG_prev=Cleared; HDG=Undefined, 19; -, {HDG_prev= Base_State_Undefined;}; HDG_prev=Undefined;
HDG_prev=Cleared; HDG=Selected, 19; -, {HDG_prev = Base_State_Selected;}; HDG_prev=Selected;
HDG_prev=Undefined; HDG=Selected, 19; -, {HDG_prev = Base_State_Selected;}; HDG_prev=Selected;
HDG_prev=Selected; HDG=Undefined, 19; -, {HDG_prev= Base_State_Undefined;}; HDG_prev=Undefined;

FD_Cues_On=False;
FD_Cues_On=Undefined; 3, ((Onside_FD == On_Off_On)); -, {FD_Cues_On = BOOLEAN_True;};
    FD_Cues_On=True;
FD_Cues_On=False; 3, ((Onside_FD == On_Off_On)); -, {FD_Cues_On = BOOLEAN_True;};
    FD_Cues_On=True;
FD_Cues_On=Undefined; 3, ((!(Onside_FD == On_Off_On))); -, {FD_Cues_On = BOOLEAN_False;};
    FD_Cues_On=False;
FD_Cues_On=True; 3, ((!(Onside_FD == On_Off_On))); -, {FD_Cues_On = BOOLEAN_False;};
    FD_Cues_On=False;

FD_Cues_On_prev=Undefined;
FD_Cues_On_prev=True; FD_Cues_On=Undefined, 19; -, {FD_Cues_On_prev = BOOLEAN_Undefined;};
    FD_Cues_On_prev=Undefined;
FD_Cues_On_prev=False; FD_Cues_On=Undefined, 19; -, {FD_Cues_On_prev = BOOLEAN_Undefined;};
    FD_Cues_On_prev=Undefined;
FD_Cues_On_prev=Undefined; FD_Cues_On=True, 19; -, {FD_Cues_On_prev = BOOLEAN_True;};
    FD_Cues_On_prev=True;
```

```
FD_Cues_On_prev=False; FD_Cues_On=True, 19; -, {FD_Cues_On_prev = BOOLEAN_True;};
    FD_Cues_On_prev=True;
FD_Cues_On_prev=Undefined; FD_Cues_On=False, 19; -, {FD_Cues_On_prev = BOOLEAN_False;};
    FD_Cues_On_prev=False;
FD_Cues_On_prev=True; FD_Cues_On=False, 19; -, {FD_Cues_On_prev = BOOLEAN_False;};
    FD_Cues_On_prev=False;

Mode_Annunciations_On=False;
Mode_Annunciations_On=Undefined; 4, ((Modes == On_Off_On));
    -, {Mode_Annunciations_On = BOOLEAN_True;}; Mode_Annunciations_On=True;
Mode_Annunciations_On=False; 4, ((Modes == On_Off_On));
    -, {Mode_Annunciations_On = BOOLEAN_True;}; Mode_Annunciations_On=True;
Mode_Annunciations_On=Undefined; 4, ((!(Modes == On_Off_On)));
    -, {Mode_Annunciations_On = BOOLEAN_False;}; Mode_Annunciations_On=False;
Mode_Annunciations_On=True; 4, ((!(Modes == On_Off_On)));
    -, {Mode_Annunciations_On = BOOLEAN_False;}; Mode_Annunciations_On=False;

Mode_Annunciations_On_prev=Undefined;
Mode_Annunciations_On_prev=True; Mode_Annunciations_On=Undefined, 19;
    -, {Mode_Annunciations_On_prev = BOOLEAN_Undefined;};
    Mode_Annunciations_On_prev=Undefined;
Mode_Annunciations_On_prev=False; Mode_Annunciations_On=Undefined, 19;
    -, {Mode_Annunciations_On_prev = BOOLEAN_Undefined;};
    Mode_Annunciations_On_prev=Undefined;
Mode_Annunciations_On_prev=Undefined; Mode_Annunciations_On=True, 19;
    -, {Mode_Annunciations_On_prev = BOOLEAN_True;};
    Mode_Annunciations_On_prev=True;
Mode_Annunciations_On_prev=False; Mode_Annunciations_On=True, 19;
    -, {Mode_Annunciations_On_prev = BOOLEAN_True;};
    Mode_Annunciations_On_prev=True;
Mode_Annunciations_On_prev=Undefined; Mode_Annunciations_On=False, 19;
    -, {Mode_Annunciations_On_prev = BOOLEAN_False;};
    Mode_Annunciations_On_prev=False;
Mode_Annunciations_On_prev=True; Mode_Annunciations_On=False, 19;
    -, {Mode_Annunciations_On_prev = BOOLEAN_False;};
    Mode_Annunciations_On_prev=False;

Is_HDG_Active=False;
Is_HDG_Active=Undefined; 5, ((HDG == Base_State_Selected));
    -, {Is_HDG_Active = BOOLEAN_True;}; Is_HDG_Active=True;
Is_HDG_Active=False; 5, ((HDG == Base_State_Selected));
    -, {Is_HDG_Active = BOOLEAN_True;}; Is_HDG_Active=True;
Is_HDG_Active=Undefined; 5, ((!(HDG == Base_State_Selected)));
    -, {Is_HDG_Active = BOOLEAN_False;}; Is_HDG_Active=False;
Is_HDG_Active=True; 5, ((!(HDG == Base_State_Selected)));
    -, {Is_HDG_Active = BOOLEAN_False;}; Is_HDG_Active=False;

Is_HDG_Active_prev=Undefined;
Is_HDG_Active_prev=True; Is_HDG_Active=Undefined, 19;
    -, {Is_HDG_Active_prev = BOOLEAN_Undefined;}; Is_HDG_Active_prev=Undefined;
Is_HDG_Active_prev=False; Is_HDG_Active=Undefined, 19;
    -, {Is_HDG_Active_prev = BOOLEAN_Undefined;}; Is_HDG_Active_prev=Undefined;
Is_HDG_Active_prev=Undefined; Is_HDG_Active=True, 19;
    -, {Is_HDG_Active_prev = BOOLEAN_True;}; Is_HDG_Active_prev=True;
Is_HDG_Active_prev=False; Is_HDG_Active=True, 19;
    -, {Is_HDG_Active_prev = BOOLEAN_True;}; Is_HDG_Active_prev=True;
Is_HDG_Active_prev=Undefined; Is_HDG_Active=False, 19;
```

```
    -, {Is_HDG_Active_prev = BOOLEAN_False;}; Is_HDG_Active_prev=False;
Is_HDG_Active_prev=True; Is_HDG_Active=False, 19;
    -, {Is_HDG_Active_prev = BOOLEAN_False;}; Is_HDG_Active_prev=False;

ROLL=Undefined;
ROLL=Cleared; 6, ((Modes == On_Off_Undefined) || (Modes != On_Off_On));
    -, {ROLL = Base_State_Undefined;}; ROLL=Undefined;
ROLL=Selected; 6, ((Modes == On_Off_Undefined) || (Modes != On_Off_On));
    -, {ROLL = Base_State_Undefined;}; ROLL=Undefined;
ROLL=Undefined; 6, ((Is_HDG_Active == 1) && (!(Modes == On_Off_On)) &&
    (Modes != On_Off_Undefined) && (Modes == On_Off_On)); -, {ROLL = Base_State_Cleared;};
    ROLL=Cleared;
ROLL=Undefined; 6, ((!(Is_HDG_Active == 1)) && (Modes == On_Off_On) &&
    (Modes != On_Off_Undefined) && (Modes == On_Off_On)); -, {ROLL = Base_State_Selected;};
    ROLL=Selected;
ROLL=Cleared; 6, ((!(Is_HDG_Active == 1)) && (Modes == On_Off_On) &&
    (Modes != On_Off_Undefined) && (Modes == On_Off_On)); -, {ROLL = Base_State_Selected;};
    ROLL=Selected;
ROLL=Selected; 6, ((!(HDG_Switch_prev == Switch_ON) && (HDG_Switch == Switch_ON)) &&
    (!(HDG_prev == Base_State_Selected)) && (Modes != On_Off_Undefined) &&
    (Modes == On_Off_On) || (!(Modes_prev == On_Off_Off) && (Modes == On_Off_Off)) &&
    (Modes != On_Off_Undefined) && (Modes == On_Off_On)); -, {ROLL = Base_State_Cleared;};
    ROLL=Cleared;


ROLL_prev=Undefined;
ROLL_prev=Selected; ROLL=Cleared, 19; -, {ROLL_prev = Base_State_Cleared;};
    ROLL_prev=Cleared;
ROLL_prev=Undefined; ROLL=Cleared, 19; -, {ROLL_prev = Base_State_Cleared;};
    ROLL_prev=Cleared;
ROLL_prev=Cleared; ROLL=Undefined, 19; -, {ROLL_prev= Base_State_Undefined;};
    ROLL_prev=Undefined;
ROLL_prev=Cleared; ROLL=Selected, 19; -, {ROLL_prev = Base_State_Selected;};
    ROLL_prev=Selected;
ROLL_prev=Undefined; ROLL=Selected, 19; -, {ROLL_prev = Base_State_Selected;};
    ROLL_prev=Selected;
ROLL_prev=Selected; ROLL=Undefined, 19; -, {ROLL_prev= Base_State_Undefined;};
    ROLL_prev=Undefined;

Is_ROLL_Selected=False;
Is_ROLL_Selected=Undefined; 7, ((ROLL == Base_State_Selected));
    -, {Is_ROLL_Selected = BOOLEAN_True;}; Is_ROLL_Selected=True;
Is_ROLL_Selected=False; 7, ((ROLL == Base_State_Selected));
    -, {Is_ROLL_Selected = BOOLEAN_True;}; Is_ROLL_Selected=True;
Is_ROLL_Selected=Undefined; 7, ((!(ROLL == Base_State_Selected)));
    -, {Is_ROLL_Selected = BOOLEAN_False;}; Is_ROLL_Selected=False;
Is_ROLL_Selected=True; 7, ((!(ROLL == Base_State_Selected)));
    -, {Is_ROLL_Selected = BOOLEAN_False;}; Is_ROLL_Selected=False;

Is_ROLL_Selected_prev=Undefined;
Is_ROLL_Selected_prev=True; Is_ROLL_Selected=Undefined, 19;
    -, {Is_ROLL_Selected_prev = BOOLEAN_Undefined;}; Is_ROLL_Selected_prev=Undefined;
Is_ROLL_Selected_prev=False; Is_ROLL_Selected=Undefined, 19;
    -, {Is_ROLL_Selected_prev = BOOLEAN_Undefined;}; Is_ROLL_Selected_prev=Undefined;
Is_ROLL_Selected_prev=Undefined; Is_ROLL_Selected=True, 19;
    -, {Is_ROLL_Selected_prev = BOOLEAN_True;}; Is_ROLL_Selected_prev=True;
Is_ROLL_Selected_prev=False; Is_ROLL_Selected=True, 19;
```

```
    -, {Is_ROLL_Selected_prev = BOOLEAN_True;}; Is_ROLL_Selected_prev=True;
Is_ROLL_Selected_prev=Undefined; Is_ROLL_Selected=False, 19;
    -, {Is_ROLL_Selected_prev = BOOLEAN_False;}; Is_ROLL_Selected_prev=False;
Is_ROLL_Selected_prev=True; Is_ROLL_Selected=False, 19;
    -, {Is_ROLL_Selected_prev = BOOLEAN_False;}; Is_ROLL_Selected_prev=False;

Is_ROLL_Active=False;
Is_ROLL_Active=Undefined; 8, ((ROLL == Base_State_Selected));
    -, {Is_ROLL_Active = BOOLEAN_True;}; Is_ROLL_Active=True;
Is_ROLL_Active=False; 8, ((ROLL == Base_State_Selected));
    -, {Is_ROLL_Active = BOOLEAN_True;}; Is_ROLL_Active=True;
Is_ROLL_Active=Undefined; 8, ((!(ROLL == Base_State_Selected)));
    -, {Is_ROLL_Active = BOOLEAN_False;}; Is_ROLL_Active=False;
Is_ROLL_Active=True; 8, ((!(ROLL == Base_State_Selected)));
    -, {Is_ROLL_Active = BOOLEAN_False;}; Is_ROLL_Active=False;

Is_ROLL_Active_prev=Undefined;
Is_ROLL_Active_prev=True; Is_ROLL_Active=Undefined, 19;
    -, {Is_ROLL_Active_prev = BOOLEAN_Undefined;}; Is_ROLL_Active_prev=Undefined;
Is_ROLL_Active_prev=False; Is_ROLL_Active=Undefined, 19;
    -, {Is_ROLL_Active_prev = BOOLEAN_Undefined;}; Is_ROLL_Active_prev=Undefined;
Is_ROLL_Active_prev=Undefined; Is_ROLL_Active=True, 19;
    -, {Is_ROLL_Active_prev = BOOLEAN_True;}; Is_ROLL_Active_prev=True;
Is_ROLL_Active_prev=False; Is_ROLL_Active=True, 19;
    -, {Is_ROLL_Active_prev = BOOLEAN_True;}; Is_ROLL_Active_prev=True;
Is_ROLL_Active_prev=Undefined; Is_ROLL_Active=False, 19;
    -, {Is_ROLL_Active_prev = BOOLEAN_False;}; Is_ROLL_Active_prev=False;
Is_ROLL_Active_prev=True; Is_ROLL_Active=False, 19;
    -, {Is_ROLL_Active_prev = BOOLEAN_False;}; Is_ROLL_Active_prev=False;

Is_HDG_Selected=False;
Is_HDG_Selected=Undefined; 9, ((HDG == Base_State_Selected));
    -, {Is_HDG_Selected = BOOLEAN_True;}; Is_HDG_Selected=True;
Is_HDG_Selected=False; 9, ((HDG == Base_State_Selected));
    -, {Is_HDG_Selected = BOOLEAN_True;}; Is_HDG_Selected=True;
Is_HDG_Selected=Undefined; 9, ((!(HDG == Base_State_Selected)));
    -, {Is_HDG_Selected = BOOLEAN_False;}; Is_HDG_Selected=False;
Is_HDG_Selected=True; 9, ((!(HDG == Base_State_Selected)));
    -, {Is_HDG_Selected = BOOLEAN_False;}; Is_HDG_Selected=False;

Is_HDG_Selected_prev=Undefined;
Is_HDG_Selected_prev=True; Is_HDG_Selected=Undefined, 19;
    -, {Is_HDG_Selected_prev = BOOLEAN_Undefined;}; Is_HDG_Selected_prev=Undefined;
Is_HDG_Selected_prev=False; Is_HDG_Selected=Undefined, 19;
    -, {Is_HDG_Selected_prev = BOOLEAN_Undefined;}; Is_HDG_Selected_prev=Undefined;
Is_HDG_Selected_prev=Undefined; Is_HDG_Selected=True, 19;
    -, {Is_HDG_Selected_prev = BOOLEAN_True;}; Is_HDG_Selected_prev=True;
Is_HDG_Selected_prev=False; Is_HDG_Selected=True, 19;
    -, {Is_HDG_Selected_prev = BOOLEAN_True;}; Is_HDG_Selected_prev=True;
Is_HDG_Selected_prev=Undefined; Is_HDG_Selected=False, 19;
    -, {Is_HDG_Selected_prev = BOOLEAN_False;}; Is_HDG_Selected_prev=False;
Is_HDG_Selected_prev=True; Is_HDG_Selected=False, 19;
    -, {Is_HDG_Selected_prev = BOOLEAN_False;}; Is_HDG_Selected_prev=False;

VS=Undefined;
VS=Cleared; 10, ((Modes == On_Off_Undefined) || (Modes != On_Off_On));
    -, {VS = Base_State_Undefined;}; VS=Undefined;
```

```
VS=Selected; 10, ((Modes == On_Off_Undefined) || (Modes != On_Off_On));
    -, {VS = Base_State_Undefined;}; VS=Undefined;
VS=Undefined; 10, ((!(!(VS_Switch_prev == Switch_ON) && (VS_Switch == Switch_ON))) &&
    (Modes != On_Off_Undefined) && (Modes == On_Off_On)); -, {VS = Base_State_Cleared;};
    VS=Cleared;
VS=Undefined; 10, ((!(VS_Switch_prev == Switch_ON) && (VS_Switch == Switch_ON)) &&
    (Modes != On_Off_Undefined) && (Modes == On_Off_On)); -, {VS = Base_State_Selected;};
    VS=Selected;
VS=Cleared; 10, ((!(VS_Switch_prev == Switch_ON) && (VS_Switch == Switch_ON)) &&
    (Modes != On_Off_Undefined) && (Modes == On_Off_On)); -, {VS = Base_State_Selected;};
    VS=Selected;
VS=Selected; 10, ((!(VS_Switch_prev == Switch_ON) && (VS_Switch == Switch_ON)) &&
    (Modes != On_Off_Undefined) && (Modes == On_Off_On) || (!(VS_Switch_prev == Switch_ON) &&
    (VS_Switch == Switch_ON)) && (!(VS_prev == Base_State_Selected)) &&
    (Modes != On_Off_Undefined) && (Modes == On_Off_On) || (!(Modes_prev == On_Off_Off) &&
    (Modes == On_Off_Off)) && (Modes != On_Off_Undefined) && (Modes == On_Off_On));
    -, {VS = Base_State_Cleared;}; VS=Cleared;


VS_prev=Undefined;
VS_prev=Selected; VS=Cleared, 19; -, {VS_prev = Base_State_Cleared;}; VS_prev=Cleared;
VS_prev=Undefined; VS=Cleared, 19; -, {VS_prev = Base_State_Cleared;}; VS_prev=Cleared;
VS_prev=Cleared; VS=Undefined, 19; -, {VS_prev= Base_State_Undefined;}; VS_prev=Undefined;
VS_prev=Cleared; VS=Selected, 19; -, {VS_prev = Base_State_Selected;}; VS_prev=Selected;
VS_prev=Undefined; VS=Selected, 19; -, {VS_prev = Base_State_Selected;}; VS_prev=Selected;
VS_prev=Selected; VS=Undefined, 19; -, {VS_prev= Base_State_Undefined;}; VS_prev=Undefined;

Is_VS_Selected=False;
Is_VS_Selected=Undefined; 11, ((VS == Base_State_Selected));
    -, {Is_VS_Selected = BOOLEAN_True;}; Is_VS_Selected=True;
Is_VS_Selected=False; 11, ((VS == Base_State_Selected));
    -, {Is_VS_Selected = BOOLEAN_True;}; Is_VS_Selected=True;
Is_VS_Selected=Undefined; 11, ((!(VS == Base_State_Selected)));
    -, {Is_VS_Selected = BOOLEAN_False;}; Is_VS_Selected=False;
Is_VS_Selected=True; 11, ((!(VS == Base_State_Selected)));
    -, {Is_VS_Selected = BOOLEAN_False;}; Is_VS_Selected=False;

Is_VS_Selected_prev=Undefined;
Is_VS_Selected_prev=True; Is_VS_Selected=Undefined, 19;
    -, {Is_VS_Selected_prev = BOOLEAN_Undefined;}; Is_VS_Selected_prev=Undefined;
Is_VS_Selected_prev=False; Is_VS_Selected=Undefined, 19;
    -, {Is_VS_Selected_prev = BOOLEAN_Undefined;}; Is_VS_Selected_prev=Undefined;
Is_VS_Selected_prev=Undefined; Is_VS_Selected=True, 19;
    -, {Is_VS_Selected_prev = BOOLEAN_True;}; Is_VS_Selected_prev=True;
Is_VS_Selected_prev=False; Is_VS_Selected=True, 19;
    -, {Is_VS_Selected_prev = BOOLEAN_True;}; Is_VS_Selected_prev=True;
Is_VS_Selected_prev=Undefined; Is_VS_Selected=False, 19;
    -, {Is_VS_Selected_prev = BOOLEAN_False;}; Is_VS_Selected_prev=False;
Is_VS_Selected_prev=True; Is_VS_Selected=False, 19;
    -, {Is_VS_Selected_prev = BOOLEAN_False;}; Is_VS_Selected_prev=False;

Is_VS_Active=False;
Is_VS_Active=Undefined; 12, ((VS == Base_State_Selected)); -, {Is_VS_Active = BOOLEAN_True;};
    Is_VS_Active=True;
Is_VS_Active=False; 12, ((VS == Base_State_Selected)); -, {Is_VS_Active = BOOLEAN_True;};
    Is_VS_Active=True;
Is_VS_Active=Undefined; 12, ((!(VS == Base_State_Selected)));
```

```
    -, {Is_VS_Active = BOOLEAN_False;}; Is_VS_Active=False;
Is_VS_Active=True; 12, ((!(VS == Base_State_Selected))); -, {Is_VS_Active = BOOLEAN_False;};
    Is_VS_Active=False;

Is_VS_Active_prev=Undefined;
Is_VS_Active_prev=True; Is_VS_Active=Undefined, 19;
    -, {Is_VS_Active_prev = BOOLEAN_Undefined;}; Is_VS_Active_prev=Undefined;
Is_VS_Active_prev=False; Is_VS_Active=Undefined, 19;
    -, {Is_VS_Active_prev = BOOLEAN_Undefined;}; Is_VS_Active_prev=Undefined;
Is_VS_Active_prev=Undefined; Is_VS_Active=True, 19;
    -, {Is_VS_Active_prev = BOOLEAN_True;}; Is_VS_Active_prev=True;
Is_VS_Active_prev=False; Is_VS_Active=True, 19;
    -, {Is_VS_Active_prev = BOOLEAN_True;}; Is_VS_Active_prev=True;
Is_VS_Active_prev=Undefined; Is_VS_Active=False, 19;
    -, {Is_VS_Active_prev = BOOLEAN_False;}; Is_VS_Active_prev=False;
Is_VS_Active_prev=True; Is_VS_Active=False, 19;
    -, {Is_VS_Active_prev = BOOLEAN_False;}; Is_VS_Active_prev=False;

PITCH=Undefined;
PITCH=Cleared; 13, ((Modes == On_Off_Undefined) || (Modes != On_Off_On));
    -, {PITCH = Base_State_Undefined;}; PITCH=Undefined;
PITCH=Selected; 13, ((Modes == On_Off_Undefined) || (Modes != On_Off_On));
    -, {PITCH = Base_State_Undefined;}; PITCH=Undefined;
PITCH=Undefined; 13, ((Is_VS_Active == 1) && (!(Modes == On_Off_On)) &&
    (Modes != On_Off_Undefined) && (Modes == On_Off_On));
    -, {PITCH = Base_State_Cleared;}; PITCH=Cleared;
PITCH=Undefined; 13, ((!(Is_VS_Active == 1)) && (Modes == On_Off_On) &&
    (Modes != On_Off_Undefined) && (Modes == On_Off_On));
    -, {PITCH = Base_State_Selected;}; PITCH=Selected;
PITCH=Cleared; 13, ((!(Is_VS_Active == 1)) && (Modes == On_Off_On) &&
    (Modes != On_Off_Undefined) && (Modes == On_Off_On));
    -, {PITCH = Base_State_Selected;}; PITCH=Selected;
PITCH=Selected; 13, ((!(VS_Switch_prev == Switch_ON) && (VS_Switch == Switch_ON)) &&
    (!(VS_prev == Base_State_Selected)) && (Modes != On_Off_Undefined) &&
    (Modes == On_Off_On) || (!(Modes_prev == On_Off_Off) && (Modes == On_Off_Off)) &&
    (Modes != On_Off_Undefined) && (Modes == On_Off_On)); -, {PITCH = Base_State_Cleared;};
    PITCH=Cleared;


PITCH_prev=Undefined;
PITCH_prev=Selected; PITCH=Cleared, 19; -, {PITCH_prev = Base_State_Cleared;};
    PITCH_prev=Cleared;
PITCH_prev=Undefined; PITCH=Cleared, 19; -, {PITCH_prev = Base_State_Cleared;};
    PITCH_prev=Cleared;
PITCH_prev=Cleared; PITCH=Undefined, 19; -, {PITCH_prev= Base_State_Undefined;};
    PITCH_prev=Undefined;
PITCH_prev=Cleared; PITCH=Selected, 19; -, {PITCH_prev = Base_State_Selected;};
    PITCH_prev=Selected;
PITCH_prev=Undefined; PITCH=Selected, 19; -, {PITCH_prev = Base_State_Selected;};
    PITCH_prev=Selected;
PITCH_prev=Selected; PITCH=Undefined, 19; -, {PITCH_prev= Base_State_Undefined;};
    PITCH_prev=Undefined;

Is_PITCH_Selected=False;
Is_PITCH_Selected=Undefined; 14, ((PITCH == Base_State_Selected));
    -, {Is_PITCH_Selected = BOOLEAN_True;}; Is_PITCH_Selected=True;
Is_PITCH_Selected=False; 14, ((PITCH == Base_State_Selected));
```

```
    -, {Is_PITCH_Selected = BOOLEAN_True;}; Is_PITCH_Selected=True;
Is_PITCH_Selected=Undefined; 14, ((!(PITCH == Base_State_Selected)));
    -, {Is_PITCH_Selected = BOOLEAN_False;}; Is_PITCH_Selected=False;
Is_PITCH_Selected=True; 14, ((!(PITCH == Base_State_Selected)));
    -, {Is_PITCH_Selected = BOOLEAN_False;}; Is_PITCH_Selected=False;

Is_PITCH_Selected_prev=Undefined;
Is_PITCH_Selected_prev=True; Is_PITCH_Selected=Undefined, 19;
    -, {Is_PITCH_Selected_prev = BOOLEAN_Undefined;}; Is_PITCH_Selected_prev=Undefined;
Is_PITCH_Selected_prev=False; Is_PITCH_Selected=Undefined, 19;
    -, {Is_PITCH_Selected_prev = BOOLEAN_Undefined;}; Is_PITCH_Selected_prev=Undefined;
Is_PITCH_Selected_prev=Undefined; Is_PITCH_Selected=True,
    19; -, {Is_PITCH_Selected_prev = BOOLEAN_True;}; Is_PITCH_Selected_prev=True;
Is_PITCH_Selected_prev=False; Is_PITCH_Selected=True, 19;
    -, {Is_PITCH_Selected_prev = BOOLEAN_True;}; Is_PITCH_Selected_prev=True;
Is_PITCH_Selected_prev=Undefined; Is_PITCH_Selected=False, 19;
    -, {Is_PITCH_Selected_prev = BOOLEAN_False;}; Is_PITCH_Selected_prev=False;
Is_PITCH_Selected_prev=True; Is_PITCH_Selected=False, 19;
    -, {Is_PITCH_Selected_prev = BOOLEAN_False;}; Is_PITCH_Selected_prev=False;

Is_PITCH_Active=False;
Is_PITCH_Active=Undefined; 15, ((PITCH == Base_State_Selected));
    -, {Is_PITCH_Active = BOOLEAN_True;}; Is_PITCH_Active=True;
Is_PITCH_Active=False; 15, ((PITCH == Base_State_Selected));
    -, {Is_PITCH_Active = BOOLEAN_True;}; Is_PITCH_Active=True;
Is_PITCH_Active=Undefined; 15, ((!(PITCH == Base_State_Selected)));
    -, {Is_PITCH_Active = BOOLEAN_False;}; Is_PITCH_Active=False;
Is_PITCH_Active=True; 15, ((!(PITCH == Base_State_Selected)));
    -, {Is_PITCH_Active = BOOLEAN_False;}; Is_PITCH_Active=False;

Is_PITCH_Active_prev=Undefined;
Is_PITCH_Active_prev=True; Is_PITCH_Active=Undefined, 19;
    -, {Is_PITCH_Active_prev = BOOLEAN_Undefined;}; Is_PITCH_Active_prev=Undefined;
Is_PITCH_Active_prev=False; Is_PITCH_Active=Undefined, 19;
    -, {Is_PITCH_Active_prev = BOOLEAN_Undefined;}; Is_PITCH_Active_prev=Undefined;
Is_PITCH_Active_prev=Undefined; Is_PITCH_Active=True, 19;
    -, {Is_PITCH_Active_prev = BOOLEAN_True;}; Is_PITCH_Active_prev=True;
Is_PITCH_Active_prev=False; Is_PITCH_Active=True, 19;
    -, {Is_PITCH_Active_prev = BOOLEAN_True;}; Is_PITCH_Active_prev=True;
Is_PITCH_Active_prev=Undefined; Is_PITCH_Active=False, 19;
    -, {Is_PITCH_Active_prev = BOOLEAN_False;}; Is_PITCH_Active_prev=False;
Is_PITCH_Active_prev=True; Is_PITCH_Active=False, 19;
    -, {Is_PITCH_Active_prev = BOOLEAN_False;}; Is_PITCH_Active_prev=False;

HDG_Lamp=OFF;
HDG_Lamp=OFF; 16, ((HDG == Base_State_Selected)); -, {HDG_Lamp = Lamp_ON;}; HDG_Lamp=ON;
HDG_Lamp=Undefined; 16, ((HDG == Base_State_Selected)); -, {HDG_Lamp = Lamp_ON;}; HDG_Lamp=ON;
HDG_Lamp=ON; 16, ((!(HDG == Base_State_Selected))); -, {HDG_Lamp = Lamp_OFF;}; HDG_Lamp=OFF;
HDG_Lamp=Undefined; 16, ((!(HDG == Base_State_Selected))); -, {HDG_Lamp = Lamp_OFF;}; HDG_Lamp=OFF;


HDG_Lamp_prev=Undefined;
HDG_Lamp_prev=ON; HDG_Lamp=OFF, 19; -, {HDG_Lamp_prev = Lamp_OFF;}; HDG_Lamp_prev=OFF;
HDG_Lamp_prev=Undefined; HDG_Lamp=OFF, 19; -, {HDG_Lamp_prev = Lamp_OFF;}; HDG_Lamp_prev=OFF;
HDG_Lamp_prev=OFF; HDG_Lamp=Undefined, 19; -, {HDG_Lamp_prev= Lamp_Undefined;};
    HDG_Lamp_prev=Undefined;
HDG_Lamp_prev=OFF; HDG_Lamp=ON, 19; -, {HDG_Lamp_prev = Lamp_ON;}; HDG_Lamp_prev=ON;
```

```
HDG_Lamp_prev=Undefined; HDG_Lamp=ON, 19; -, {HDG_Lamp_prev = Lamp_ON;}; HDG_Lamp_prev=ON;
HDG_Lamp_prev=ON; HDG_Lamp=Undefined, 19; -, {HDG_Lamp_prev= Lamp_Undefined;};
    HDG_Lamp_prev=Undefined;

VS_Lamp=OFF;
VS_Lamp=OFF; 17, ((VS == Base_State_Selected)); -, {VS_Lamp = Lamp_ON;}; VS_Lamp=ON;
VS_Lamp=Undefined; 17, ((VS == Base_State_Selected)); -, {VS_Lamp = Lamp_ON;}; VS_Lamp=ON;
VS_Lamp=ON; 17, ((!(VS == Base_State_Selected))); -, {VS_Lamp = Lamp_OFF;}; VS_Lamp=OFF;
VS_Lamp=Undefined; 17, ((!(VS == Base_State_Selected))); -, {VS_Lamp = Lamp_OFF;}; VS_Lamp=OFF;


VS_Lamp_prev=Undefined;
VS_Lamp_prev=ON; VS_Lamp=OFF, 19; -, {VS_Lamp_prev = Lamp_OFF;}; VS_Lamp_prev=OFF;
VS_Lamp_prev=Undefined; VS_Lamp=OFF, 19; -, {VS_Lamp_prev = Lamp_OFF;}; VS_Lamp_prev=OFF;
VS_Lamp_prev=OFF; VS_Lamp=Undefined, 19; -, {VS_Lamp_prev= Lamp_Undefined;};
    VS_Lamp_prev=Undefined;
VS_Lamp_prev=OFF; VS_Lamp=ON, 19; -, {VS_Lamp_prev = Lamp_ON;}; VS_Lamp_prev=ON;
VS_Lamp_prev=Undefined; VS_Lamp=ON, 19; -, {VS_Lamp_prev = Lamp_ON;}; VS_Lamp_prev=ON;
VS_Lamp_prev=ON; VS_Lamp=Undefined, 19; -, {VS_Lamp_prev= Lamp_Undefined;};
    VS_Lamp_prev=Undefined;

FD_Switch=Undefined;
FD_Switch=OFF; 20; -, {FD_Switch = Switch_OFF;}; FD_Switch=OFF;
FD_Switch=OFF; 20; -, {FD_Switch = Switch_ON;}; FD_Switch=ON;
FD_Switch=Undefined; 20; -, {FD_Switch = Switch_OFF;}; FD_Switch=OFF;
FD_Switch=OFF; 20; -, {FD_Switch = Switch_Undefined;}; FD_Switch=Undefined;
FD_Switch=ON; 20; -, {FD_Switch = Switch_OFF;}; FD_Switch=OFF;
FD_Switch=ON; 20; -, {FD_Switch = Switch_ON;}; FD_Switch=ON;
FD_Switch=Undefined; 20; -, {FD_Switch = Switch_ON;}; FD_Switch=ON;
FD_Switch=ON; 20; -, {FD_Switch = Switch_Undefined;}; FD_Switch=Undefined;
FD_Switch=Undefined; 20; -, {FD_Switch = Switch_Undefined;}; FD_Switch=Undefined;

FD_Switch_prev=Undefined;
FD_Switch_prev=OFF; FD_Switch=Undefined, 19; -, {FD_Switch_prev = Switch_Undefined;};
    FD_Switch_prev=Undefined;
FD_Switch_prev=Undefined; FD_Switch=OFF, 19; -, {FD_Switch_prev = Switch_OFF;};
    FD_Switch_prev=OFF;
FD_Switch_prev=OFF; FD_Switch=ON, 19; -, {FD_Switch_prev = Switch_ON;}; FD_Switch_prev=ON;
FD_Switch_prev=ON; FD_Switch=Undefined, 19; -, {FD_Switch_prev = Switch_Undefined;};
    FD_Switch_prev=Undefined;
FD_Switch_prev=Undefined; FD_Switch=ON, 19; -, {FD_Switch_prev = Switch_ON;};
    FD_Switch_prev=ON;
FD_Switch_prev=ON; FD_Switch=OFF, 19; -, {FD_Switch_prev = Switch_OFF;}; FD_Switch_prev=OFF;

HDG_Switch=Undefined;
HDG_Switch=OFF; 20; -, {HDG_Switch = Switch_OFF;}; HDG_Switch=OFF;
HDG_Switch=OFF; 20; -, {HDG_Switch = Switch_ON;}; HDG_Switch=ON;
HDG_Switch=Undefined; 20; -, {HDG_Switch = Switch_OFF;}; HDG_Switch=OFF;
HDG_Switch=OFF; 20; -, {HDG_Switch = Switch_Undefined;}; HDG_Switch=Undefined;
HDG_Switch=ON; 20; -, {HDG_Switch = Switch_OFF;}; HDG_Switch=OFF;
HDG_Switch=ON; 20; -, {HDG_Switch = Switch_ON;}; HDG_Switch=ON;
HDG_Switch=Undefined; 20; -, {HDG_Switch = Switch_ON;}; HDG_Switch=ON;
HDG_Switch=ON; 20; -, {HDG_Switch = Switch_Undefined;}; HDG_Switch=Undefined;
HDG_Switch=Undefined; 20; -, {HDG_Switch = Switch_Undefined;}; HDG_Switch=Undefined;

HDG_Switch_prev=Undefined;
HDG_Switch_prev=OFF; HDG_Switch=Undefined, 19; -, {HDG_Switch_prev = Switch_Undefined;};
```

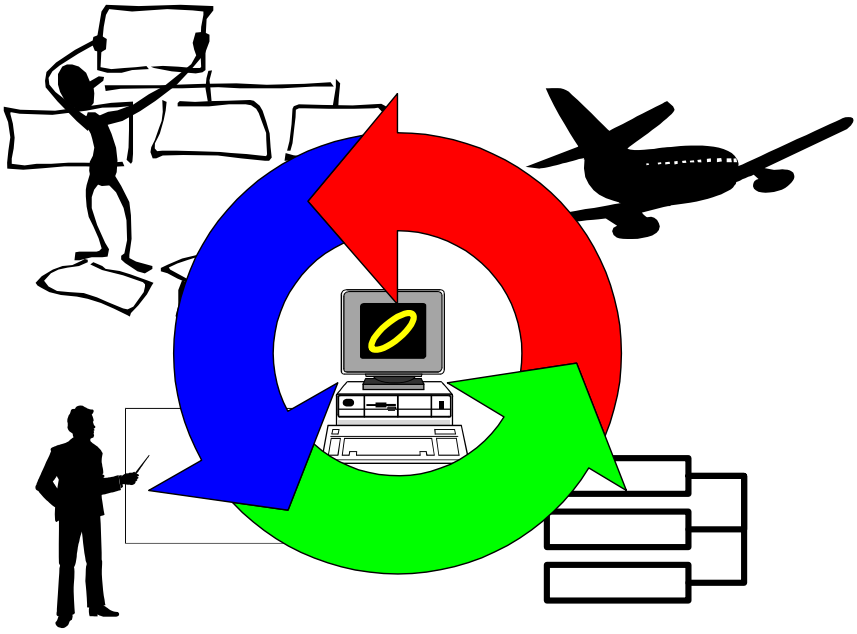
```
    HDG_Switch_prev=Undefined;
HDG_Switch_prev=Undefined; HDG_Switch=OFF, 19; -, {HDG_Switch_prev = Switch_OFF;};
    HDG_Switch_prev=OFF;
HDG_Switch_prev=OFF; HDG_Switch=ON, 19; -, {HDG_Switch_prev = Switch_ON;}; HDG_Switch_prev=ON;
HDG_Switch_prev=ON; HDG_Switch=Undefined, 19; -, {HDG_Switch_prev = Switch_Undefined;};
    HDG_Switch_prev=Undefined;
HDG_Switch_prev=Undefined; HDG_Switch=ON, 19; -, {HDG_Switch_prev = Switch_ON;};
    HDG_Switch_prev=ON;
HDG_Switch_prev=ON; HDG_Switch=OFF, 19; -, {HDG_Switch_prev = Switch_OFF;};
    HDG_Switch_prev=OFF;

VS_Switch=Undefined;
VS_Switch=OFF; 20; -, {VS_Switch = Switch_OFF;}; VS_Switch=OFF;
VS_Switch=OFF; 20; -, {VS_Switch = Switch_ON;}; VS_Switch=ON;
VS_Switch=Undefined; 20; -, {VS_Switch = Switch_OFF;}; VS_Switch=OFF;
VS_Switch=OFF; 20; -, {VS_Switch = Switch_Undefined;}; VS_Switch=Undefined;
VS_Switch=ON; 20; -, {VS_Switch = Switch_OFF;}; VS_Switch=OFF;
VS_Switch=ON; 20; -, {VS_Switch = Switch_ON;}; VS_Switch=ON;
VS_Switch=Undefined; 20; -, {VS_Switch = Switch_ON;}; VS_Switch=ON;
VS_Switch=ON; 20; -, {VS_Switch = Switch_Undefined;}; VS_Switch=Undefined;
VS_Switch=Undefined; 20; -, {VS_Switch = Switch_Undefined;}; VS_Switch=Undefined;

VS_Switch_prev=Undefined;
VS_Switch_prev=OFF; VS_Switch=Undefined, 19; -, {VS_Switch_prev = Switch_Undefined;};
    VS_Switch_prev=Undefined;
VS_Switch_prev=Undefined; VS_Switch=OFF, 19; -, {VS_Switch_prev = Switch_OFF;};
    VS_Switch_prev=OFF;
VS_Switch_prev=OFF; VS_Switch=ON, 19; -, {VS_Switch_prev = Switch_ON;}; VS_Switch_prev=ON;
VS_Switch_prev=ON; VS_Switch=Undefined, 19; -, {VS_Switch_prev = Switch_Undefined;};
    VS_Switch_prev=Undefined;
VS_Switch_prev=Undefined; VS_Switch=ON, 19; -, {VS_Switch_prev = Switch_ON;};
    VS_Switch_prev=ON;
VS_Switch_prev=ON; VS_Switch=OFF, 19; -, {VS_Switch_prev = Switch_OFF;}; VS_Switch_prev=OFF;

0;
0; -; -; 1;
1; -; -; 2;
2; -; -; 3;
3; -; -; 4;
4; -; -; 5;
5; -; -; 6;
6; -; -; 7;
7; -; -; 8;
8; -; -; 9;
9; -; -; 10;
10; -; -; 11;
11; -; -; 12;
12; -; -; 13;
13; -; -; 14;
14; -; -; 15;
15; -; -; 16;
16; -; -; 17;
17; -; -; 18;
18; -; -; 19;
19; -; -; 20;
20; -; -; 0;
```

Appendix B

RSML^{-e} and NIMBUS

The NIMBUS Environment for Specification of Safety Critical Systems DRAFT

For the RSML^{-e} Specification Language



Developed by

The Critical Systems Research Group (CriSys)
Department of Computer Science and Engineering
University of Minnesota
Minneapolis, Minnesota

Manual By
Mike W. Whalen, Mats P.E. Heimdahl, and Jeffrey M. Thompson,

Table of Contents

1 Introduction and Document Overview

The Critical Systems Research Group originally developed the *NIMBUS* environment for safety critical systems at the University of Minnesota. The environment provided a framework for the development of software for safety critical systems, including simulation, code generation, and visualization. It supported the RSML (Requirements State Machine Language) formalism. In cooperation with Safeware Engineering Corporation, the existing tools were modified to support the RSML^{-e} language and expanded to include a graphical user interface for the simulation engine, a construction tool for the specifications, and implementation of analysis.

This document contains information necessary to be productive with the NIMBUS environment.

1.1 Constructing Specifications

Specifications in the Nimbus environment are simple ASCII test files that can be created with any text editor. We are currently working on support for creating RSML^{-e} specifications in the WinEdt editor. This support, however, is not available at this time. When the specification is ready, it is readable by the NimbusSim tool and can be simulated or analyzed. The following document describes the way in which specifications are constructed in the Nimbus environment.

- **RSML^{-e} Language Manual:** The language manual provides a basic introduction to the syntax and semantics of the RSML-e language. Through the use of a simple example, the pump control system, the reader will gain an understanding of the features of the language and how they can be used.

1.2 Simulating and Analyzing Specifications

The NimbusSim tools provide the Nimbus environment with the ability to simulate and analyze specifications. The NimbusSim application itself provides the ability to load and graphically display specifications. Furthermore, the analyst can execute and analyze the specification. In addition to the NimbusSim simulator, the Nimbus environment provides for inter-process communication to be used for input and output of the simulations. This feature is supported by several applications: the NimbusManager and COM objects for integration to C++ and Visual Basic, and other associated clients. This framework allows the RSML^{-e} specification to be executed with a variety of other components that accurately model the environment of the controller. All this is described in the simulation and analysis documentation:

- **NimbusSim Graphical User Interface Manual:** This document describes the features of the NimbusSim graphical user interface menus, toolbars, and command window. It discusses how to load specifications, layout the state hierarchy, simulate specifications, connect channels, and perform analysis.
- **NimbusChannel Users Guide:** This guide describes how NimbusSim can be connected to other applications via the NimbusChannel framework built on top of Microsoft's Component Object Model (COM). It discusses both which clients are currently available and how to create clients in C++ or Visual Basic using the provided COM objects and wrapper classes.

1.3 Examples

Examples are an important part of learning any new specification language. Research is ongoing at the University of Minnesota and elsewhere into how to structure and construct specifications in the RSML^{-e} language. Even so, we provide here one completely worked-out example showing various features of the language and how they can be used.

- **The Clean Room:** Consider a room that is supposed to be sealed at all times. To enter the room you have to go through an airlock. To get in, you have to open the front door, step into the air-lock, close the door, open the inside door, step into the room, and finally close the inside door. This simple example will be used to illustrate many concepts from the RSML^{-e} language.

Additional examples will be added as time progresses.

1.4 Technical Documentation and Research Results

Any documentation for the RSML^{-e} language and its associated tools would be incomplete without a discussion of the formal semantics (a major feature of the language) as well as the research results that led to the creation of RSML^{-e}.

These documents are currently available from the Critical Systems research groups at the University of Minnesota.

2 Getting Started with NIMBUS

NIMBUS is a powerful and flexible framework for the construction of specifications for safety-critical systems that is, nonetheless, easy to use. To get started, read the release notes provided with the installation files. These notes contain important information about late-breaking features, compatibility and installation issues, and more.

2.1 System Requirements

The NIMBUS environment runs under Windows 95, Windows 98, Windows NT 4.0, and Windows 2000. To take full advantage of the environment, users should have the following software (not provided by the University of Minnesota) installed on their workstations:

- Microsoft Internet Explorer 4.0 or higher (required by HTML help)

Nevertheless, NIMBUS can be used without this additional software; the user will simply not be able to utilize the features of the HTML format help (HTML help can be browsed with any web browser).

2.2 Installation and Setup Instructions

2.2.1 Acquiring the Installation Files

The installation files for the NIMBUS environment are downloadable from the Critical Systems Research Group's web site at the following URL. You must obtain a username and password from either the Critical Systems Research Group to successfully download the distribution. The URL for the download is:

http://www.cs.umn.edu/crisys/NIMBUS/download

Follow the instructions given on the page to complete the download of the NIMBUS Environment installation files.

2.2.2 Installing the Tools

To install the tools, follow the simple steps below:

1. Download the installation file from the web (or from the CD)
2. Unzip the installation file into a temporary directory.
3. Double click on the NIMBUS Installer program file
4. Follow the instructions through the installation

Important Note: *If you already have a previous version of* NIMBUS *installed on your machine, you must first uninstall* NIMBUS *before you can attempt a new install. We do not know the source of the problems, but the installation program (provided by Microsoft) tends to freeze up or exhibits other kinds of unexpected behaviors if installing over an existing version is attempted. Thus,* ***uninstall first and then install a new version****.*

2.3 Tools of the NIMBUS Environment

The NIMBUS environment encompasses a number of different tools. Thus, a number of items are contained in the start menu program group that is created by the setup program. The paragraphs below describe the tools that are available in the environment, their function, and where the documentation about their features can be located.

2.3.1 NimbusSim

NIMBUSSim is the full-featured simulation and analysis tool of the NIMBUS framework. It allows you to perform simulation and analysis on RSML^{-e} specifications, copy and paste specification layouts, produce testing scripts and more. NIMBUSSim can be accessed with the NIMBUSSim icon from the program group created by setup.

Documentation of NIMBUSSim can be found in the following places:

- Application specific topics are found in the online help
- The NIMBUSSim Graphical User Interface Manual

2.3.2 NIMBUS Manager

The NIMBUS Manager is the application that allows you to control the connections between various components in the NIMBUSChannel communication environment. The NIMBUSManager allows you to connect and disconnect sources and destinations so that the correct systems configuration can be achieved. The NIMBUS Manager can be accessed with the NIMBUS Manager icon from the program group created by setup. Documentation of the NIMBUS Manager can be found in the NIMBUSChannel Users Guide.

2.3.3 NIMBUSChannel Client

The NIMBUSChannel Client is a simple client application written in C++ that allows the user to connect to the NIMBUSChannel framework and act as a source, destination, or observer on a channel. The NIMBUSChannel client can be accessed through the program group created by setup by selecting the NIMBUSChannel Client icon. Documentation for the NIMBUSChannel Client can be found in the NIMBUSChannel Users Guide.

3 Introduction to RSML^{-e}

RSML^{-e} is a finite-state machine based specification language that represents the evolution of the specification language RSML (Requirements State Machine Language). RSML was designed during the specification of TCAS II (Traffic Alert and Collision Avoidance System II) by the Safety research group at the University of California, Irvine. RSML was based on Statecharts, including such features as the event propagation mechanism and the notion of hierarchical state machines.

RSML^{-e}, a refinement and improvement of RSML, was created for a number of reasons. Among them were the error prone use of the event propagation mechanism and the difficulty of reusability in RSML specifications.

The purpose of RSML^{-e}, and the associated tools in the *NIMBUS* environment, is to allow analysts to specify safety critical systems with high reliability. RSML^{-e} contributes to this goal by being a readable specification language that is usable and understandable by all stakeholders in a specification effort. Therefore, the language is suitable for manual inspections and reviews. Nevertheless, RSML^{-e} is a fully formal specification language; thus, analysts can also perform formal analysis and simulation on the requirement model. To achieve a high level of confidence, all three approaches (manual inspections, formal analysis, and simulation) must be used in concert. This is enabled in the NIMBUS environment by the RSML^{-e} language and its associated tool set.

An RSML^{-e} specification consists of a collection of variables, the next state relations for the variables, functions, macros, constants, and interfaces. Variables describe the internal state of the system. Interfaces describe how the specification interacts with the external environment. Functions and Macros are mechanisms for representing common expressions and predicates, respectively, in order to make specifications more concise. These constructs are discussed in the following sections with the context of a simple example: the clean room.

The clean room specification is as follows:

> *Consider a room that is supposed to be sealed at all times. To enter the room you have to go through an airlock. To get in, you have to open the front door, step into the air-lock, close the door, open the inside door, step into the room, and finally close the inside door. To open a door, a person must request the door using some means (e.g. a button). Only one person should be allowed in the airlock at a time, and if the airlock is in use, other requests should be denied until the airlock is unoccupied. At no point should both doors to the airlock be open, unless a power failure or catastrophic event occurs. If both doors are open, then the clean room must be considered contaminated, with serious financial consequences.*
>
> *When entering the clean room, an individual must be "cleaned" using air scrubbers to remove particles from their clothes. The duration for this cleaning is some application-*

defined constant. Until the individual is clean, they should not be allowed into the clean room.

The system shall provide two alarm features. If an airlock is occupied for longer than a specified duration, a timeout alarm should be generated. Also, in case of some system malfunction or other catastrophe, pressing buttons within the clean room and the airlock will generate a panic alarm. In this event, both doors should be unlocked and people should be able to leave the clean room unhindered. If an alarm is generated, it continues until an administrator resets the system.

The clean room may have 1-*n* airlocks, all of which behave identically.

In this manual, we have made a few simplifying assumptions to the clean room problem:

- The clean room only contains one airlock
- It is the administrator's responsibility to ensure that the system is in a consistent state when the system is reset (i.e. no people in the airlocks)
- The sensors/actuators do not malfunction
- The cleaning interval is 60 seconds; the timeout interval is 5 minutes

Here are the system inputs/outputs.

Input:	Description:
panic_button: bool	panic_button is true when any of the panic buttons are pressed
reset_button: bool	reset_button is true when a system reset request is generated
inner_door_request: bool	inner_door_request is true in the duration when a user is requesting to exit the clean room
outer_door_request: bool	outer_door_request is true in the duration when a user is requesting to enter the clean room
inner_door_open: bool	inner_door_open is true when the inner door is open
outer_door_open: bool	outer_door_open is true when the outer door is open
airlock_occupied: bool	airlock_occupied is true when the airlock is occupied (could be through a floor or motion sensor)
clock: bool	A clock pulse that is issued once per second

Output:	Description:
inner_door_lock	inner_door_lock is true when the inner door of the airlock is locked
outer_door_lock	outer_door_lock is true when the outer door of the airlock is locked
decontaminate: bool	decontaminate is true during the decontamination interval when the system should "scrub" a user who is entering the clean room
panic_alarm: bool	panic_alarm is true at the instant when a user presses the panic_button, and true thereafter until the reset_button has been pressed

timeout_alarm: bool	timeout_alarm is true at the instant when the user has been in an airlock for longer than *300* seconds

Once these inputs are set, it is pretty straightforward to describe the procedure for entering/exiting the airlock:

1. Initially, both doors to the airlock are locked.
2. The user requests to enter (exit) the airlock
   - If the airlock is 'in-use' the request is denied
   - Otherwise, unlock the outer (inner) door
3. The user opens the outer (inner) door
4. The user closes the outer (inner) door
   - If the airlock is occupied, then lock the outer (inner) door and proceed to the next stage
   - If the airlock is unoccupied, the user must have decided not to enter the airlock, so the airlock is no longer 'in-use'
5. If the user is entering, then clean the user for 60 seconds
6. Unlock the inner (outer) door and wait for the user to exit
7. The user opens the door
8. The user closes the door
   - If the airlock is still occupied, then the user must not yet have exited; repeat
   - If the airlock is unoccupied, then the process is complete and the airlock is no longer 'in-use'.

3.1 Synchronous Languages

The semantics of RSML^{-e} puts it in a class of languages called synchronous languages. Since this class of languages may be largely unknown to the novice RSML^{-e} user, we here include a short introduction.

Synchronous languages were proposed as a software engineering tool in the late 1970s and independently in the programming language community in the late 1980s as a technique for modeling and constructing reactive (i.e. process control) systems. As these systems often control the behavior of several cooperating machines, they are often most easily modeled as a set of cooperating concurrent tasks. Synchronous languages provide primitives that allow programmers to naturally model this concurrent structure, and also to consider that their programs react *instantaneously* to external events [Halbwachs91]. By using this *synchrony hypothesis*, and enforcing certain constraints within a synchronous language, it is possible to create logically concurrent programs that are deterministic both in terms of functionality and time.

There are two main styles of synchronous languages: imperative and dataflow. Imperative synchronous languages are quite similar to standard imperative languages like C or Ada, but are augmented with constructs to deal with process instantiation, communication, and termination. Unlike similar constructs in, for example, Ada, these constructs are considered logically instantaneous, and are essential to creating deterministic programs. The leading example of this style is the programming language Esterel[Berry00].

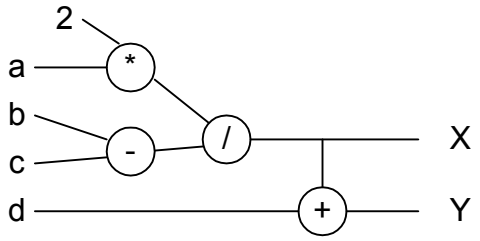
In contrast, dataflow languages are inspired by control theory. Many of the engineers who design reactive systems model their systems as networks of operators transforming flows of data, and at a higher level by block diagrams that group these networks into reusable components. Dataflow languages allow these models to directly realize the software control system.

As the basis of a high-level programming language, the dataflow model has several merits:

- It is a completely functional model without side effects. This feature makes the model well suited to formal verification and program transformation. It also facilitates reuse, as a module will behave the same way in any context into which it is embedded.
- It is a naturally parallel model, in which the only constraints on parallelism are enforced by the data-dependencies between variables. This allows for parallel implementations to be realized, either in software, or directly in hardware.

Dataflow models can be either *synchronous* or *asynchronous*. In an asynchronous dataflow model, the outputs of the system are continually recomputed depending on the inputs to the system. In the synchronous model, however, real-time is broken into a sequence of instants in which the model is recomputed. The synchronous model is better suited to translation into a programming language, as it more naturally matches the behavior of a computer program. Therefore, all of the dataflow-style languages adopt some form of this approach.

As an example, consider a system that computes the values of two variables, X and Y, based on 4 inputs: a, b, c, and d:



(a)

X = 2a / (b - c)
Y = X + d

(b)

Figure 1: A dataflow model and its associated set of equations.

This diagram is to be read left-to-right, with the inputs "flowing" through the system of operators to create the outputs at the right side. The diagram can be represented more concisely as a set of equations, as shown at right. We name the inputs to the dataflow model *input variables* and all variables that are computed by the model *state variables.*

The variables in a dataflow model are used to label a particular computation graph; they are not used as constraints. Therefore, it is incorrect to view the equations as a set of constraints on the model: a set of equations such as {X = 2a/Y, Y = X + d} does not correspond to an operator network because X and Y mutually refer to one another. Such a system may have no solution or infinitely many solutions, so cannot be directly used as a deterministic program. If viewed as a graph, these sets of equations have *data dependency cycles*, and are considered incorrect.

However, in order for the language to be useful, we must be able to have mutual reference between variables. To allow benign cyclic dependencies, a *delay operator* is added. The operator returns the value of an expression, delayed one instant. For example: {X = 2a / Y, Y = delay(X, 1) + d} defines a system where X is equal to 2a divided by the current value of Y, while Y is equal to the *previous* value of X plus the current value of d. The second parameter of the delay operator defines the value of the operator at the initial instant, when the previous value of X is undefined. Systems of equations of this form always have a single solution. The delay operator is also the mechanism for recording state about the model. For example, we can construct a counter over the natural numbers by simply defining the set of equations {x := delay(x+1, 0)}.

Finally, some notion of *selection* is added to assignment expressions. Depending on the language, this notion can be realized as an if/then statement, a series of cases, or a set of transitions. From these elements, at its core, a dataflow program can be viewed as simply a set of input variables and assignment equations of the form $\{X_0 = E_0, X_1 = E_1, ..., X_n = E_n\}$ that must be acyclic in terms of data dependencies.

RSML^{-e} is a synchronous data-flow language of the structure described above.

3.2 Variables and States Variables

Variables are central to RSML^{-e}. An RSML^{-e} specification is constructed from variables organized in parallel or hierarchically (this will be explained in more detail below). We make a slight distinction between variables and state variables. State variables in RSML^{-e} are simply variables that have graphical representation that will show up in the GUI during execution. In the remainder of this report we will use variables and state variables interchangeably—they are both referring to some variable in the model that is used to capture the system state.

In other specification languages, states are often used to represent activities of the controller. This is a subtle, yet important, difference; the *state* of the system may or may not have anything to do with which particular *activities* the system is performing at any particular moment. Therefore, in RSML^{-e} states can be thought of as more of a hierarchical enumerated type and are closer to traditional finite state machines than those found in other specification languages.

Variable assignments in RSML^{-e} govern how the state variables can change from one value to another—they capture the next state relation. There are two styles of variable assignments—the transition style and the assignment style. Both styles will be illustrated below.

3.3 Types

The various variables in an RSML^{-e} specification must have a type. Thus, the types needed to model a system are often determined early during modeling. In this case, we will model open and closed doors, alarms that may be on or off, and doors that can be locked or unlocked. For the clean room, we start with the types below

```
TYPE_DEF on_off         { off, on }
TYPE_DEF door_status    { closed, open }
TYPE_DEF door_lock_status { unlocked, locked }
TYPE_DEF button_status  { not_pressed, pressed }
```

Type definitions in RSML^{-e} simply declare a user defined enumerated type. Note here that all types in RSML^{-e} contain an implicit value UNDEFINED. This value is used when we simply do not know what the value of a variable may be. This situation occurs, for example, at startup, when no input arrives, or when the variable is not relevant (the part of the variable hierarchy where it resided is not uses at the time). The issue of UNDEFINED will be revisited later in this manual.

The grammar for type definitions is very simple.

```
type_def              : TYPE_DEF IDENTIFIER '{' enum_element_list '}'
                      ;

enum_element_list       : IDENTIFIER
                        | enum_element_list ',' IDENTIFIER
                        ;
```

3.4 Variables

When a variable is declared, we have to give it a type. This is achieved through a type reference or an implicit type declaration.

```
type_ref                : IDENTIFIER
                        | INTEGER_TYPE
                        | REAL_TYPE
                        | BOOLEAN_TYPE
                        | TIME
                        ;
```

The clean room specification is straightforward to implement in RSML^{-e}. Using the RSML^{-e} hierarchy constructs (discussed below), it is possible to directly visualize and control the relationships between the variables that describe the status of the airlock:

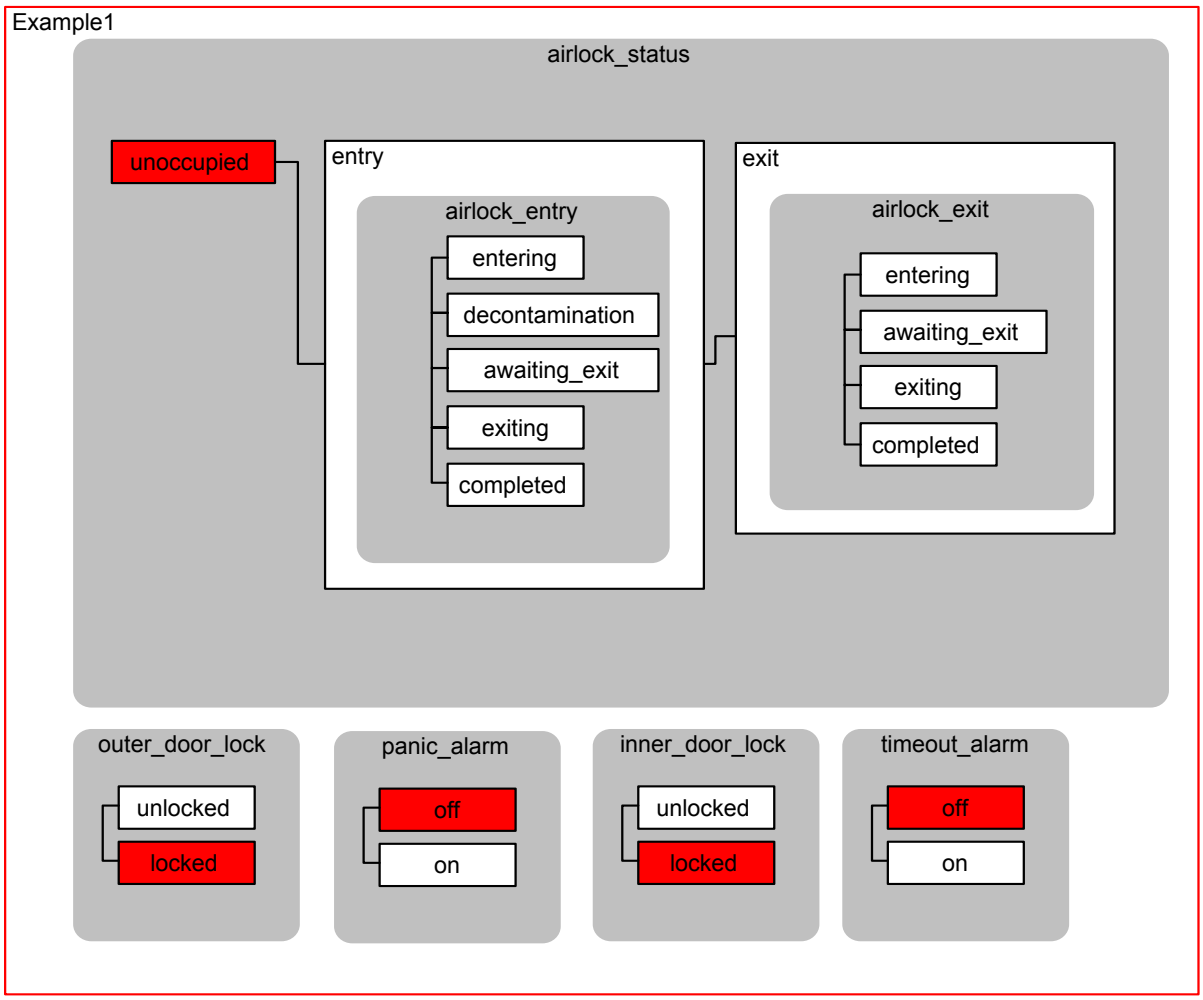


Figure 2: NIMBUS visualization of clean room RSML^{-e} specification

At the top level, we have an *airlock_status* variable, which describes whether or not the airlock is occupied, along with variables detailing the status of alarms and door locks. Under the *airlock_status* variable, we place the variables describing the process of entering and exiting the airlock. These variables are only relevant when *airlock_status* is in entry or exit mode, respectively.

This view of the state variables is the one produced by the NIMBUSSim RSML^{-e} simulator. It shows each variable value as a box with black text and a white background (this is the default in the tool—it can be customized by the user). The possible values of a variable are joined with connecting lines. Note, however, that these lines do not necessarily indicate that it is possible for the state variable to change from any of its values to any other of its values (see below, about assignment relations). ***Note:*** *This visualization may change since CriSys are not quite satisfied with the information content*. The name of state variable appears above the connected states. State variables can be nested (as shown by the *airlock_status* variable).

When an RSML^{-e} specification is loaded into the NIMBUSSim simulator, the simulator needs to assign some initial values to the state variables. This initial value is determined by which states are marked with the INITIAL_VALUE keyword in the proceeding state definitions. Thus, Figure 2 above shows the state hierarchy for the pump controller in the default configuration.

The collection of all the active states in the state hierarchy can be thought of as the configuration of the machine. The behavior of the specification is determined, in part, by the way that the state hierarchy changes from one configuration to another. These changes in configurations are determined by the definition of transitions from one state to another. Transitions are specified one state variable at a time.

Consider the behavior of the *airlock_status* state variable. This state variable is intended to capture the state of the airlock (is a person entering the room, exiting the room, or is the lock unoccupied). The definition of the state variable can be seen in Figure 3 below.

```
STATE_VARIABLE airlock_status :
   VALUES : {unoccupied, entry, exit}
   PARENT : None
   INITIAL_VALUE : unoccupied
   CLASSIFICATION: State

   Transition unoccupied TO exit IF inner_door_request

   Transition unoccupied TO entry IF
      TABLE
         outer_door_request            : T;
         inner_door_request            : F;
      END TABLE

   Transition exit TO unoccupied IF
      TABLE
         PREV_STEP(..airlock_exit) IN_STATE completed    : T *;
         reset_button                                    : * T;
      END TABLE

   Transition entry TO unoccupied IF
      TABLE
         PREV_STEP(..airlock_entry) IN_STATE completed   : T *;
         reset_button                                    : * T;
      END TABLE

END STATE_VARIABLE
```

Figure 3: The definition of the airlock_status state variable.

The definition includes the name of the state variable, airlock_status and the possible values the variable can take on (note that all variables can take on the value UNDEFINED in addition to

any values defined in the variable definition—this will be discussed in detail later). The next thing in the definition is the *parent* of the variable. This is the construct that allows us to organize variables in a hierarchy. For example, the variable airlock_entry has the parent airlock_status.entry (the value entry of airlock_status). The variable airlock_entry has no parent (it is on the top level) and this is indicated by the parent None. The CLASSIFICATION field of the variable definition is intended for suture extensions of the language. Currently, the field is ignored unless it states that this is a State variable. All variables classified as State variables will be visualized in the graphical user interface. Next in the definition, the possible values for the state variable are given as well as the conditions that must hold for the state variable to assume the value. Essentially, each one of the statements represents a transition from one state (state variable value) to another state. Note that some of the conditions in the specification are presented in a DNF form called And/Or tables.

During their work on the TCAS specification, the Irvine group discovered that the traditional predicate logic statements traditionally used to capture the conditions on transitions did not scale well to the large, complex expressions in TCAS. To combat this issue, they developed And/Or tables. The tables contain a column of predicates. To the right of the column of predicates are one or more columns of truth-values. During evaluation (execution of the specification), the predicates at the left will have some values. For a column in the table to be TRUE, the values of the predicates must "match up" with the truth-values indicated in one the columns (i.e., the predicate must the TRUE if there is a T in the column and FALSE if there is an F). A * in the column denotes that we don't care about the value of that particular predicate with respect to the column. For an And/Or table to be TRUE, one of the columns must completely match (i.e., be TRUE).

The conditions in the airlock_status state variable are relatively simple. Inner_door_request is a Boolean input variable that is used as a condition. We can also use abbreviations of more complex conditions (called macros) and various expressions involving integer and real variables, and time. A complete discussion of all the expressions allowed in RSML^{-e} appears later in the document.

In some instances we may want a state variable to take on a specific value *independently* of what value it had before—in essence we want transitions from ANY state to a certain state. In RSML^{-e} we achieve this with a slightly different variant of the transition definition. As can be seen in Figure 4, we are allowed to state that a variable *assumes* a value when a condition is true (independently of what value it had previously).

```
STATE_VARIABLE inner_door_lock : door_lock_status
   PARENT: NONE
   INITIAL_VALUE : locked
   CLASSIFICATION : State

   EQUALS unlocked IF inner_door_unlocked()
   EQUALS locked IF !inner_door_unlocked()
END STATE_VARIABLE
```

Figure 4: The EQUALS style of transition definition.

The grammar for state variable definitions is given below.

```
state_variable_def      : STATE_VARIABLE IDENTIFIER array_decl ':'
variable_type_decl
                            PARENT        ':' parent_decl
                            INITIAL_VALUE ':' expression
                            variable_numeric_decl
                            classification_def
                            case_list
                          END STATE_VARIABLE
                        ;

variable_numeric_decl   : /* empty */
                        | UNITS         ':' IDENTIFIER
                          EXPECTED_MIN  ':' expression
                          EXPECTED_MAX  ':' expression

variable_type_decl      : type_ref
                        | VALUES    ':' '{' enum_element_list '}'


array_decl              : /* empty */
                        | '[' expression TO expression ']'
                        ;

parent_decl             : NONE
                        | parent_name_path
                        ;

parent_name_path        : IDENTIFIER
                        | parent_name_path '.' IDENTIFIER
                        ;

classification_def      : /* empty */
                        | CLASSIFICATION ':' IDENTIFIER
                        ;

type_ref                : IDENTIFIER
                        | INTEGER_TYPE
                        | REAL_TYPE
                        | BOOLEAN_TYPE
                        | TIME
                        ;

condition               :  TABLE
                           row_list
                           END TABLE
```

```
                        |  expression /* Must return BOOLEAN */
                        ;

row_list                :  expression ':' truth_value_list ';'
                        |  row_list  expression ':' truth_value_list
';'
                        ;

truth_value_list        :  truth_value
                        |  truth_value truth_value_list
                        ;

truth_value             : 'T'
                        | 'F'
                        | '.'
                        | '*'
```

3.5 Input Variables

Input variables in RSML^{-e} allow you to model information about the environment of the system. For example, you might like to capture the inputs from sensors. The input variables are conceptually slightly different than the variables we discussed in the previous section—input variables can only be set by an *interface* when an interaction with the environment occurs. There is not notion of an output variable in RSML^{-e}; any variable can be used to output information from a model. We do, however, encourage the use of dedicated variables that act as output variables. Variables (both input variables discussed here and the 'normal' variables discussed in the previous section) can be any one of the types that are present in the RSML^{-e} type system: floating point, integer, time, or enumerated.

A type of a variable (its possible values) can be declared separately in RSML^{-e}, or it can be declared directly with the variable itself. The latter is called using an *anonymous type* since the type is never given a name and cannot be used in any other place of the specification. The other alternative is to declare a type explicitly and then use the type name when we declare the type of a variable (Figure 4).

An input variable consists of an initial value, and (for non-enumerated type variables) an expected minimum, maximum, and units. The definition of the panic_button from the clean room is shown in Figure 5. The type of the variable is Boolean. The definition of the panic_button does not require an expected minimum, expected maximum, or units since it is Boolean. Only variables of numeric types (integer and real) require those fields. There are no numeric variables in the clean room example—an example of a numeric variable from another specification (an avionics system) is included in Figure 6 for illustration.

```
IN_VARIABLE panic_button : boolean
   INITIAL_VALUE : FALSE
   CLASSIFICATION: MONITORED
END IN_VARIABLE
```

Figure 5: Boolean variable definition.

```
IN_VARIABLE Altitude : INTEGER
   INITIAL_VALUE : Undefined
   UNITS : ft
   EXPECTED_MIN : 0
   EXPECTED_MAX : 40000
END IN_VARIABLE
```

Figure 6: An integer variable definition.

3.6 Interfaces

Interfaces define a number of properties related to how the specification can interact with its environment. In RSML^{-e} the interaction with the environment is achieved by providing or consuming messages over a communication channel. Since RSML^{-e} is a synchronous language, we make the assumption that only one message is received at any point in time and that the message can be completely processed before another one arrives. If we want to model that many inputs can change simultaneously, we can put them all in the same message—the fields in a message can all change at the same time, but the fields in different messages cannot. Thus, RSML^{-e} operates under a one-message assumption. An example of a message definition in the clean room is shown in Figure 7. A different message from an avionics application is shown in Figure 8.

```
MESSAGE Update_Message {
   f_panic_button IS boolean,
   f_reset_button IS boolean,
   f_inner_door_request IS boolean,
   f_outer_door_request IS boolean,
   f_inner_door_open IS boolean,
   f_outer_door_open IS boolean,
   f_airlock_occupied IS boolean
}
```

Figure 7: Message definition in the clean room.

```
MESSAGE AltitudeMessage {
   Alt IS INTEGER,
   aq IS AltitudeQualityType
}
```

Figure 8: Message definition for altitude.

First, the interface defines the type of communication: Send-Receive or Publish-Read. Send-Receive communication is equivalent to message passing scheme. Publish-Read communication, on the other hand, buffers the message on the channel so that a message that is published might be read several times by the reader. Second, interfaces define the properties of the communication, for example the expected minimum and maximum separation between messages over the channel. Finally, the interfaces regulate the assignment of inputs to the specification to the input variables. Using this feature, simple safety and liveness constraints can be imposed by the interfaces without considering the (potentially complex) function represented by the state machine and associated definitions.

The interface for the clean room is given below (Figure 9). The interface definition begins with a declaration of the name for this interface. The simulator uses this name to hook the interface to actual communications channels from the environment. Next, the expected minimum and maximum separation for messages over the channel is given. The final part of the header for the input interface is the input action. This determines (1) the type of communication and (2) which message will be received/read on the input channel.

```
IN_INTERFACE Update_Interface :
      MIN_SEP : UNDEFINED
      MAX_SEP : UNDEFINED
      INPUT_ACTION : RECEIVE(Update_Message)
      HANDLER :
            CONDITION : TRUE
            ASSIGNMENT
                  panic_button         := f_panic_button,
                  reset_button         := f_reset_button,
                  inner_door_request   := f_inner_door_request,
                  outer_door_request   := f_outer_door_request,
                  inner_door_open      := f_inner_door_open,
                  outer_door_open      := f_outer_door_open,
                  airlock_occupied     := f_airlock_occupied
            END ASSIGNMENT
      END HANDLER
END IN_INTERFACE
```

Figure 9: Interface definition for the clean room.

The final section of the input interface consists of a number of *handlers*. Handlers are similar to transitions in a state variable. They have a condition, under which they will be executed. If the condition is TRUE, then the handler will execute and the assignments will occur. In the case in

Figure 9, the interface only has one handler and that handler will always be used (the guarding condition for the handler is simply true). A more complex interface from an avionics example is included in Figure 10.

```
IN_INTERFACE AltitudeMessageInterface :
   MIN_SEP : 50 MS
   MAX_SEP : 100 MS
   INPUT_ACTION : RECEIVE(AltitudeMessage)

RECEIVE_HANDLER :
      CONDITION :
        TABLE
        Alt <= Altitude::EXPECTED_MAX : T;
        Alt >= Altitude::EXPECTED_MIN : T;
        END TABLE
      ASSIGNMENT
         Altitude := Alt,
         AltitudeQuality := aq
      END ASSIGNMENT
   END HANDLER

RECEIVE_HANDLER :
      CONDITION :
        TABLE
         Alt <= Altitude::EXPECTED_MAX : F *;
         Alt >= Altitude::EXPECTED_MIN : * F;
        END TABLE
      ASSIGNMENT
         Altitude := UNDEFINED,
         AltitudeQuality := Bad
      END ASSIGNMENT
   END HANDLER

END IN_INTERFACE
```

Figure 10: More complex interface example from avionics.

The BNF grammar for input interfaces follows:

```
in_interface_def        : IN_INTERFACE IDENTIFIER ':'
                              MIN_SEP ':' expression
                              MAX_SEP ':' expression
                              INPUT_ACTION ':' in_interface_type_spec
'(' IDENTIFIER ')'
                              in_handler_list
                          END IN_INTERFACE
                        ;

in_interface_type_spec  : RECEIVE
                        | READ
                        ;

in_handler_list         : in_handler
                        | in_handler in_handler_list
                        ;

in_handler              : in_handler_type ':'
                              CONDITION ':' condition
                              in_assignment
                          END HANDLER
                        ;

in_handler_type         : RECEIVE_HANDLER
                        | HANDLER
                        ;

in_assignment           : /* empty */
                        | ASSIGNMENT
                           in_assignment_list
                          END ASSIGNMENT
                        ;

in_assignment_list      : identifier_name_path ASSIGN_TOKEN expression
                        | in_assignment_list ',' identifier_name_path
ASSIGN_TOKEN expression
                        ;
```

Output interfaces are similar to input interfaces. An example of an output interface from the pump controller is given in Figure 11. Again, a more complex example from avionics is included in Figure 12.

```
OUT_INTERFACE Actuator_Interface :
      MIN_SEP : UNDEFINED
      MAX_SEP : UNDEFINED
      OUTPUT_ACTION : SEND(Actuator_Message)
      HANDLER :
            CONDITION : TRUE
            ASSIGNMENT
         f_inner_door_lock    := inner_door_lock,
         f_outer_door_lock    := outer_door_lock,
         f_decontaminate      := decontaminate,
         f_panic_alarm        := panic_alarm,
         f_timeout_alarm      := timeout_alarm
            END ASSIGNMENT
      ACTION : SEND
      END HANDLER
END OUT_INTERFACE
```

Figure 11: Simple output interface.

```
OUT_INTERFACE FaultDetectionInterface :
   MIN_SEP : 50 MS
   MAX_SEP : 200 MS

   OUTPUT_ACTION : SEND(FaultMessage)

   HANDLER :
      CONDITION :
         TABLE
            ASWOpModes IN_STATE OK                  : T * ;
            ASWOpModes IN_STATE FailureDetected     : * T;
         END TABLE

      ASSIGNMENT
         fault := FaultDetectedVariable
      END ASSIGNMENT

      ACTION : SEND
   END HANDLER
END OUT_INTERFACE
```

Figure 12: More complex output interface from the avionics domain.

Finally, we provide the BNF grammar for the output interfaces.

```
out_interface_def       : OUT_INTERFACE IDENTIFIER ':'
                              MIN_SEP ':' expression
                              MAX_SEP ':' expression
                              OUTPUT_ACTION ':'
out_interface_type_spec '(' IDENTIFIER ')'
                              output_handler_list
                          END OUT_INTERFACE
                        ;

out_interface_type_spec : SEND
                        | PUBLISH
                        ;

output_handler_list     : output_handler
                        | output_handler_list output_handler
                        ;

output_handler          : HANDLER ':'
                              CONDITION ':' condition
                              out_assignment
                              ACTION ':' out_handler_type
                          END HANDLER
                        ;

out_handler_type        : SEND
                        | PUBLISH
                        | NONE

out_assignment          : /* empty */
                        | ASSIGNMENT
                           out_assignment_list
                          END ASSIGNMENT
                        ;

out_assignment_list     : IDENTIFIER ASSIGN_TOKEN expression
                        | IDENTIFIER ASSIGN_TOKEN expression ','
                          out_assignment_list
                        ;
```

3.7 Expressions in RSML^{-e}

This section provides a look at the expressions available in RSML^{-e} and their meaning. RSML^{-e} supports the standard arithmetic expressions (addition, subtraction, multiplication, and division) comparison operators (greater than, greater or equal, equal (=), not equal (!=), less than, less than or equal) as well as parenthesis for expression grouping and traditional logical not for Boolean expressions (NOT *expression*). These expressions can contain references to variables, constants, macros, and functions.

Literal values are allowed in RSML^{-e}. Floating point, integer and Boolean literals are given as expected. Enumerated literals are given as *type_name*::*enumeration_name*. Time values are given in the format *n* [H | M | S | MS] where *n* is an integer and H, M, S, MS stand for hours, minutes, seconds and milliseconds respectively. Any number of these clauses can be given separated by white space and an optional "AND;" thus "3 H 5 M" and "4 H 5 M and 3 S" are both valid time literals.

Also supported is a set of RSML^{-e} specific expressions involving, for example, previous values of variables and time. These expressions are detailed in Table 1 on page 77. The format of the entries of the table is in a peusdo-BNF grammar style. Parts that appear in italics are references to other language definitions or expressions. Parts which appear in square brackets [] are either optional or represent a choice of several values. For example [a] means that part “a” is optional and [a|b] means choose between “a” and “b.”

Expression	**Meaning**
variable_name::[EXPECTED_MIN |EXPECTED_MAX	Equal to the expected minimum or maximum of the referenced variable
PREV_VALUE(*variable_name* [, *n*])	The *n*th previous value of the variable referenced before the current step started. The *n* is optional. For example, PREV_VALUE(x) is the value that the variable x had before it took on the value that it had at the beginning of current step.
PREV_STEP(*variable_name* [, *n*])	The value that the variable had in the *n*th previous step. The *n* is optional. For example, PREV_STEP(x) is the value that variable x had at the end of the previous step (and the beginning of this step).
TIME	The current system time.
TIME(*variable_name)*	The time when the variable acquired the current value.
PREV_ASSIGN(*variable_name* [, *n*])	The time when the referenced variable acquired its *n*th previous value.

TIME_CHANGED (*variable_name* [, *n*])	The time when the state variable given by *variable_name* changed value. Note that this time may be different than when the variable last got assigned (it could have been assigned the same value).
interface_name::LAST_IO	The time that the interface referenced last performed I/O on the channel.
[*interface_name*::] [MIN_SEP | MAX_SEP]	The minimum or maximum separation for the given interface.
expression EQ_ONE_OF *expression_list*	True if the *expression* is equal to one of the comma separated expressions in the *expression_list.*
ASSIGNED(*variable_name)*	True if the variable was assigned in this step.
CHANGED(*variable_name)*	True if the variable changed its value in this step.
AT_TRUE(*expression*)	True if the expression changed from false to true in the current step.
AT_FALSE(*expression*)	True if the expression changed from true to false in the current step.
AT_CHANGED(*expression*)	True if the expression changed from one truth-value to another truth-value in the current step.

Table 1: Expressions in RSML^{-e}.

3.8 Macros and Functions

The macros and functions of RSML^{-e} complete the language by allowing the analyst to define commonly used computations in a modular way. This facilitates good structure and maintainability in the specification. Macros in RSML^{-e} are simply a compact method of writing Boolean functions. Macros are represented as an And/Or table (thus, in a way, a macro is simply a named And/Or table). A sample macro from the clean room can be seen in Figure 13. A function from the avionics domain is shown in Figure 14.

```
MACRO inner_door_unlocked() :
   TABLE
      ..airlock_entry IN_ONE_OF {awaiting_exit, exiting}    : T * *;
      ..airlock_exit IN_STATE  entering                     : * T *;
      panic_alarm = on                                      : * * T;
   END TABLE

END MACRO
```

Figure 13: Macro example.

```
TYPE_DEF Selected_Nav_Types { FMS, VOR, LOC }

FUNCTION Selected_Nav_Type(): Selected_Nav_Types
    EQUALS FMS  IF Is_Selected_Nav_Source_FMS()

    EQUALS VOR  IF
        TABLE
        VNR_Signal_Type = VOR       : T;
        Is_Selected_Nav_Source_VNR()    : T;
        END TABLE

    EQUALS LOC  IF
        TABLE
        VNR_Signal_Type = LOC       : T;
        Is_Selected_Nav_Source_VNR()    : T;
        END TABLE
END FUNCTION
```

Figure 14: Function example.

In RSML^{-e}, both Macros and functions can take parameters. Parameters allow the macros and functions to be even more modular and allow the analyst to reuse common conditions in various situations in the specification. For example, you might wish to have a macro to do pair wise sensor failure analysis for an array of sensors. The macro would contain the conditions for determining failure given two sensor inputs and the parameters would by the sensor inputs. The clean room specification does not contain any parameterized macros or functions, so no example from that specification is possible here; however, the function definition in Figure 15 illustrates the simple maximum function.

```
FUNCTION Max(a IS INTEGER, b IS INTEGER): INTEGER
    EQUALS a  IF a > b
    EQUALS b  IF a <= b
END FUNCTION
```

Figure 15: Parameterized function.

The BNF for macros and functions follow below:

```
/*------------ Macro definitions -------------------------------*/

optional_formal_parms   : /* EMPTY */
                        | '(' formal_parameter_list ')'

macro_def               : MACRO IDENTIFIER optional_formal_parms ':'
                              condition
                          END MACRO
                        ;



/*------------ Function definitions ----------------------------*/
optional_expr_list      : /* empty */
                        | expression_list

function_def            : FUNCTION IDENTIFIER '('
formal_parameter_list ')' ':' type_ref
                              case_list
                          END FUNCTION

                        | STUB_FUNCTION IDENTIFIER '('
formal_parameter_list ')' ':' type_ref
                              optional_expr_list
                          END STUB_FUNCTION
                        ;

case_list               : /* EMPTY */
                        | case_list case
                        ;

case                    : EQUALS expression IF condition
                        | TRANSITION expression TO expression IF
condition
                        ;

actual_parameter_list   : /* empty */
                        | expression_list
                        ;

formal_parameter_list   : /* empty */
                        | IDENTIFIER IS type_ref
                        | formal_parameter_list ',' IDENTIFIER IS
type_ref
                        ;
```

3.9 Advanced Language Issues

RSML^{-e} is based on the idea that the various language definitions in the specification constitute a *mathematical relation* from inputs to outputs. The relation is constructed by computing the values of the various language items (state, input, and output variables, macros, and functions). This is an important concept in the formalism of RSML^{-e} because it allows the opportunity to create various completeness and consistency criteria for the relation. However, this view of the semantics leads to a certain behavior of specifications written in the notation that might not be readily apparent to the newcomer. These are briefly discussed in the sections below. For more information, please refer to the formal semantics of RSML^{-e}.

3.9.1 Circular Dependencies

When an entity in the specification changes, the semantics of the RSML^{-e} language dictate that all entities that depend on the one that changed must be recomputed. Thus, the order of computation in the specification is determined by the data dependencies of the various language entities. This means that there can be no circular dependencies, because there would be no way to tell when to stop updating the values in the specification. That is, the data dependencies for all entities in the specification must form a directed acyclic graph.

Using expressions that reference the previous values of variables, however, will not cause circular dependencies. That is because this previous information is fixed at the start of the computation and remains constant throughout. Thus, formally, these previous values can be viewed as parts of the system state (or as additional inputs to the specification).

3.9.2 Transition Issues and Equivalence Class Evaluation

The semantics of RSML^{-e} states that for each state variable there should be a maximum of one transition taken in each step between its immediate children. A side effect of these is that the state of the machine is recomputed on a variable-by-variable basis, not on a transition-by-transition basis. The data dependencies for a variable are computed by taking the union of the data dependencies of the transitions between the values of the variable (all the expressions in the variable definition are taken into account). Furthermore, variables with parent sate values are defined to be data dependent on their parent variables in the variable hierarchy so that variables that are “higher” in the tree get evaluated before the child variables.

3.9.3 Receive Handlers

Usually, entities in the RSML^{-e} specification are recomputed when some other entity changes. However, the handlers in the RECEIVE interfaces are a special case because, in general, they should only be recomputed when a message was actually received regardless of the data dependencies of the conditions in the handlers. Therefore, to denote this separate behavior those handlers are marked with **Receive_Handler** instead of just **Handler**.

3.9.4 PREV_VALUE and PREV_STEP

The PREV_VALUE and PREV_STEP expressions in RSML^{-e} allow the user to access the previous values of variables and interfaces. This *does not* change the view of the specification as a function. Rather, these values are (necessarily) static and thus are simply viewed as additional parts of the system state.

4 NIMBUSSim Graphical User Interface

4.1 The NIMBUSSim Interface

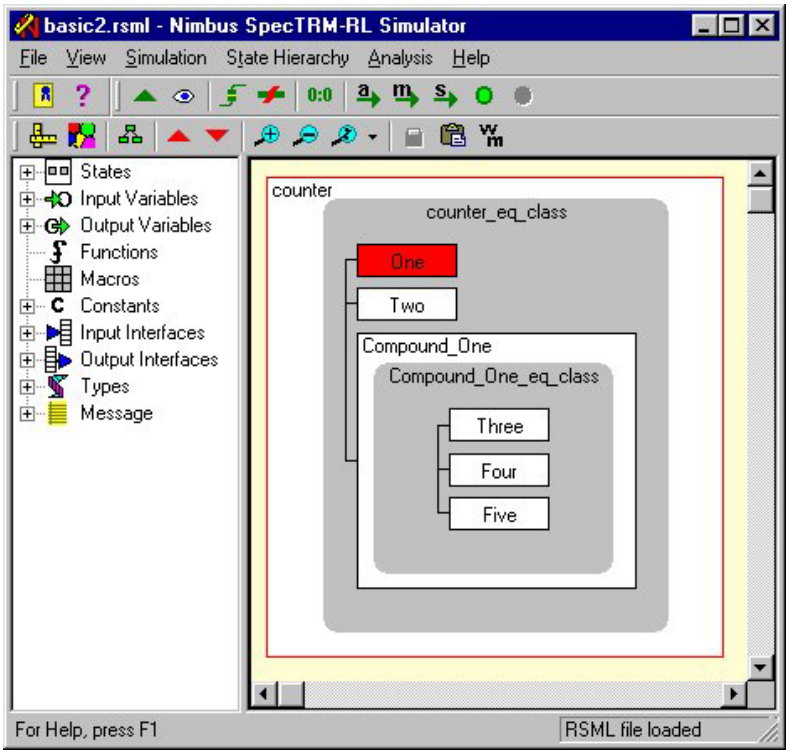


Figure 16: The NIMBUSSim GUI Main Window

4.2 Overview

The NIMBUSSim Graphical User Interface provides accurate and fast access to the functionality of the NIMBUS simulator. Specifications are visualized in two ways: a tree-structure of detailed information about the data, and a State Hierarchy Diagram that enhances the context of state relationships. Furthermore, the user may observe the effects of execution with the run-time information provided by Active State Highlighting, Variable Watch updates, and the Clock display

Launching **NIMBUSSim** causes two windows to appear: the Graphical User Interface (Fig. 1) and the Command-Line Interpreter. There exists one simulator and these two windows are two methods for issuing commands. The user may switch between them at any time.

In the next few sections we present an overview of the NIMBUS commands that are accessible through the GUI. The first section links the GUI commands with their visual appearance. The

next section summarizes the steps required to run a simulation, as they pertain to the User Interface. Finally, the Command Reference summarizes the GUI and Command Line counterparts, with some additional actions that are unique to each.

4.3 Details

4.3.1 File Menu Options

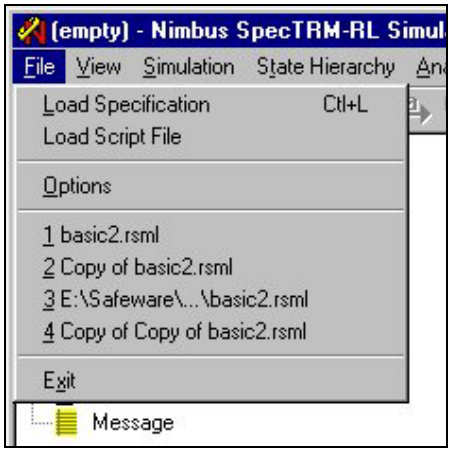


Figure 17: The File Menu

Load Specification

Opens an existing specification causing it to be loaded into the simulator. Upon successful parsing, the state hierarchy is diagrammed and specification details are loaded into the Tree View.

Load Script File

Opens a script file and loads it into the simulator for immediate evaluation.

Options

Here you can access many of the user-configurable properties from the System Options dialog. In addition, most of these options are available through their associated toolbars and menus.

Exit

Exits the simulator and interface. If there is a simulation in progress, then you must Stop the Full Execution.

4.3.2 System Options

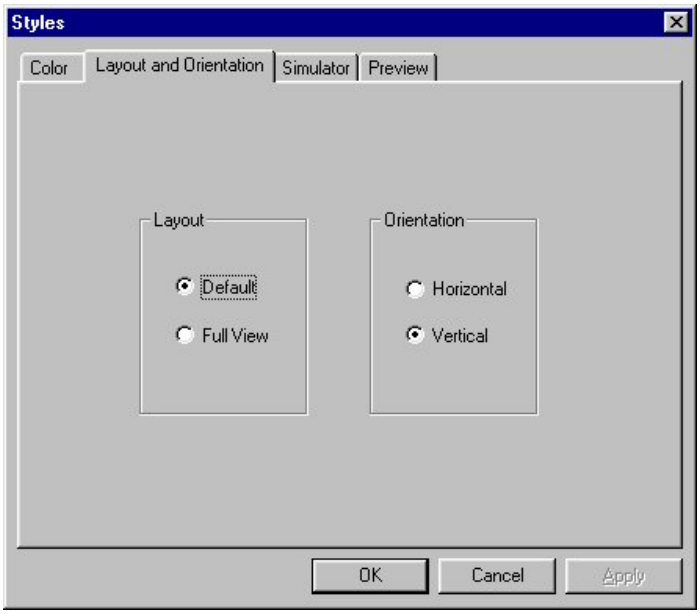


Figure 18 Layout Options

Layout options include orientation. Choose **Horizontal** for one that favors wide layouts. Choose **Vertical** for one that favors taller layouts. The Vertical layout is more useful for transferring the image to a word processor.

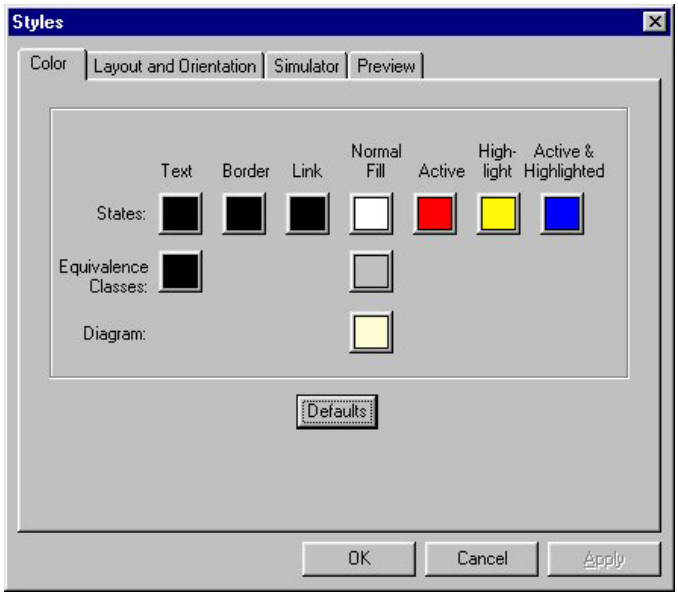


Figure 19 Color Options for State Hierarchy

You may select the State Hierarchy Diagram colors.

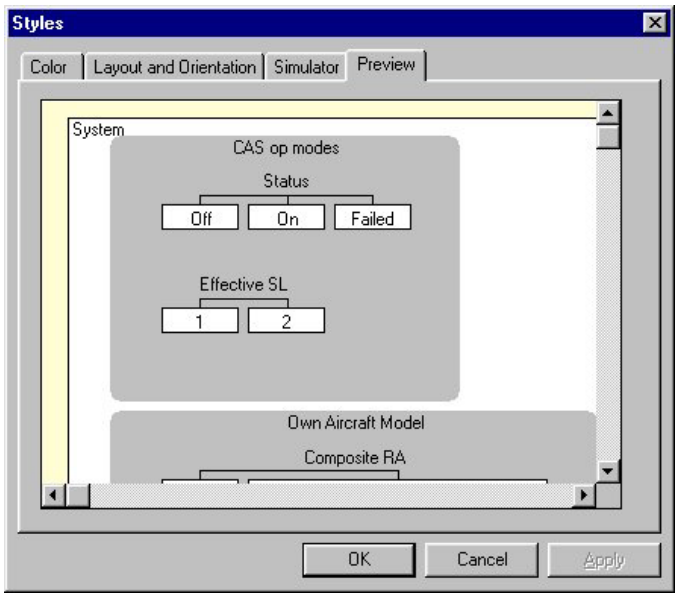


Figure 20 Preview Panel for State Hierarchy Options

You can preview the Layout, Orientation and Color configurations before committing to them .

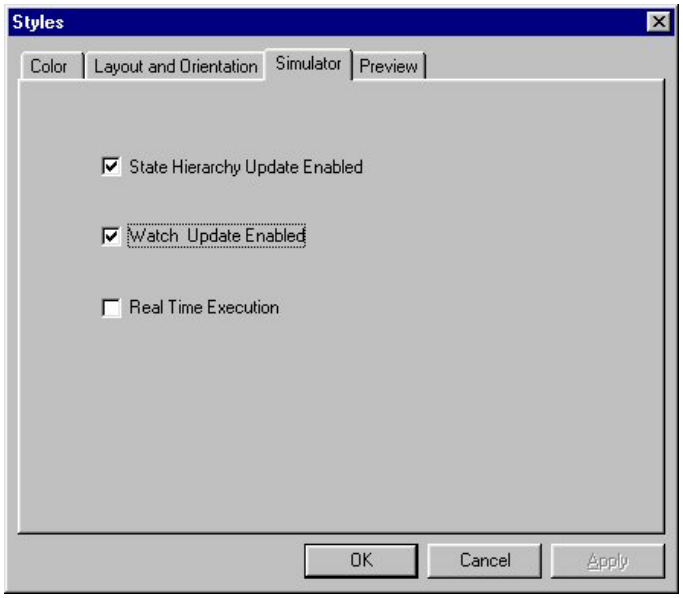


Figure 21 Simulator Options

State Diagram Updating

Check or uncheck to either enable or disable the updating of the State Hierarchy Diagram during any run command executions.

Watch Window Updating

Check or uncheck to enable or disable the updating of the Watch Window during any run command executions.

Real Time

Set the simulation clock type to either Real Time (checked) or Simulation Time (unchecked).

4.3.3 View Menu

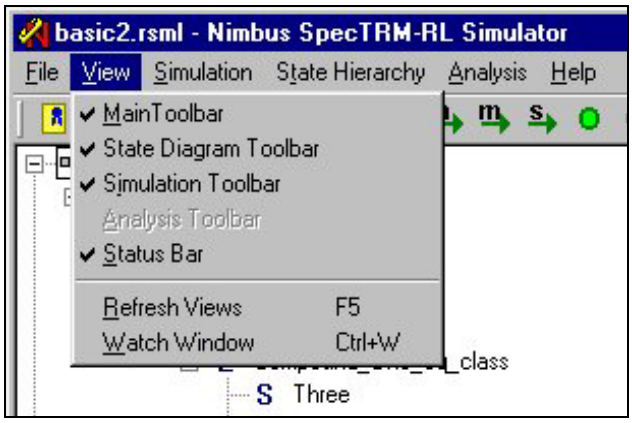


Figure 22 The View Menu

Main Toolbar

Show (checked) or hide (unchecked) the Main Toolbar. This toolbar contains commands: Load Specification, Help

State Diagram Toolbar

Show (checked) or hide (unchecked) the State Diagram Toolbar. This toolbar contains commands: Load Specification, Help

Simulation Toolbar

Show (checked) or hide (unchecked) the Simulation Toolbar.

Analysis Toolbar

Not implemented in this version of the user-interface.

Status Bar

Show (checked) or hide (unchecked) the Status Bar. This bar contains: Command Prompts, System Status, Simulation Time, Clock Type, and the State Diagram Updating, Watch Updating states.

Refresh Views

Updates the Tree, State Hierarchy, and Watch Window. This is useful for updating the State Diagram and Watch Window when their dynamic updating have been disabled (via the Simulation Menu) and you wish to view the current state of the system.

Watch Window

Shows the Watch Window.

4.3.4 Simulation Menu and Toolbar

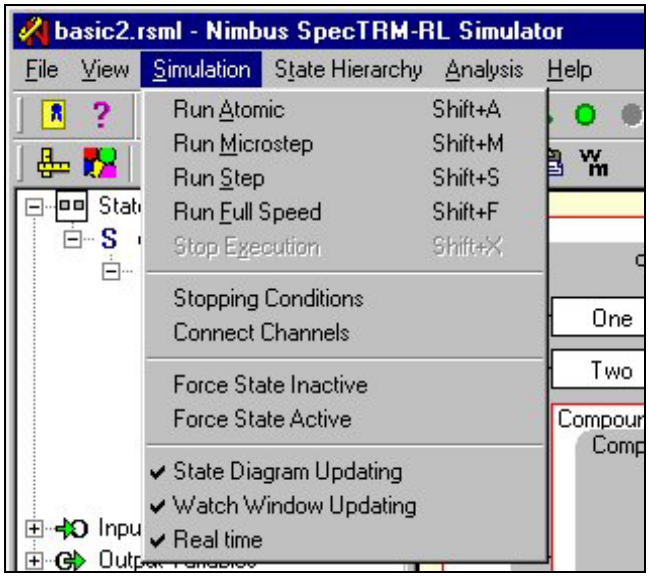


Figure 23 The Simulation Menu

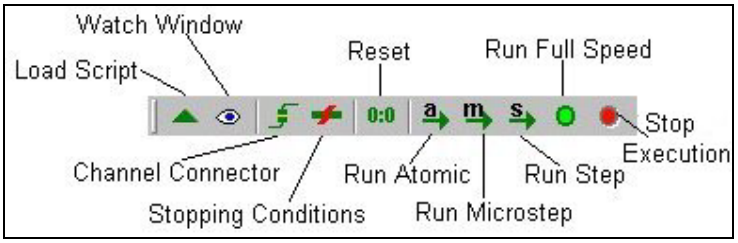


Figure 24 The Simulation Toolbar

Run Atomic

Runs one step in the active simulation.

Run Microstep

Runs one microstep in the active simulation.

Run Step

Runs one step in the active simulation.

Run Full Speed

Runs the active simulation at full speed.

Stop Execution

Stops the running simulation.

Stopping Conditions

Activates the Stopping Conditions dialog box wherein you may View, Add, or Remove Stopping Conditions.

Connect Channels

Activates the Channel Connector dialog box.

Force State Active

After the selection of a state in the State Diagram, you may force it to be *Active* here and also in a popup menu in the Diagram.

Force State Inactive

After the selection of a state in the State Diagram, you may force it to be *Inactive* here and also in a popup menu in the Diagram.

State Diagram Updating

Check or uncheck to either enable or disable the updating of the State Hierarchy Diagram during any run command executions.

Watch Window Updating

Check or uncheck to enable or disable the updating of the Watch Window during any run command executions.

Real Time

Set the simulation clock state to either Real Time (checked) or Simulation Time (unchecked).

4.3.5 State Hierarchy Menu and Toolbar

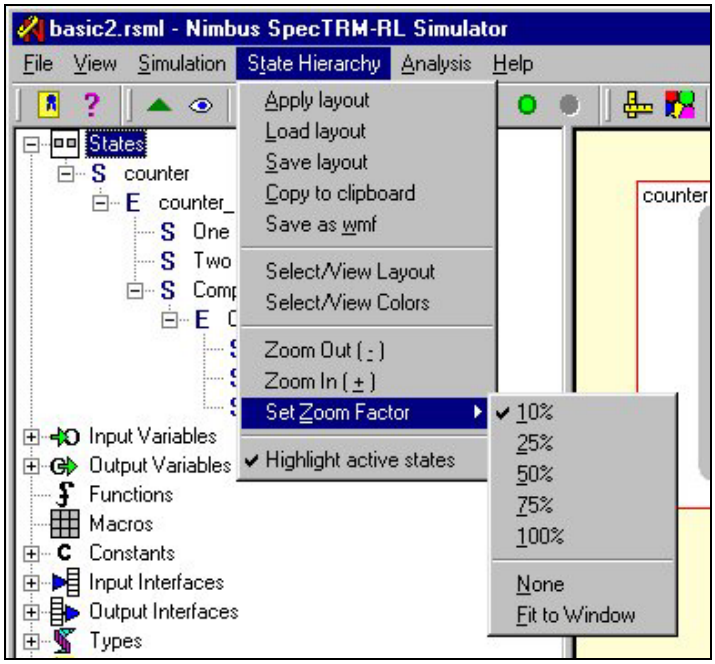


Figure 25 The State Hierarchy Menu

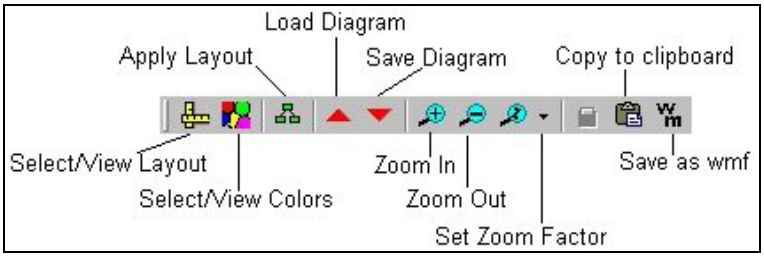


Figure 26 The State Hierarchy Toolbar

Apply layout

Apply the currently selected layout to the active specification. If you have made any modifications to a diagram and wish to apply the layout, then be certain to save your work via **Save Layout** if you wish to retain it; there is no reminder mechanism.

Load layout

You may load any layouts provided that they were created with this application's tool. The extension **af** is appended to your saved files to help identify them from within the file dialog boxes. If you load a layout that is inconsistent with the current specification, then NIMBUS will attempt to repair your graph. This feature can be used during incremental development of your system.

Save layout

Saves the current layout of the State Hierarchy Diagram. It is required that you save the diagram in a separate file from the **rsml** specification.

Copy to clipboard

Copies the Diagram layout to the Windows clipboard. This is useful for pasting the image into other applications such as word processors and image editors.

Save as wmf

Saves the layout as a Windows Metafile with a **wmf** extension. This is useful for saving the diagram in a portable format. WMF files can be inserted into numerous applications.

Select/View Layout

This option presents you with the Layout and Orientation Options page in the System Options dialog.

Select/View Colors

This option presents you with the Color Options page in the system options dialog.

Zoom Options

The zoom feature works as follows. The user selects a zoom factor. This factor is used for both zooming in and zooming out. The user can, at any time, reset to the original size by selecting **None** (see below).

Zoom Out (+)

Zoom out using the current factor.

Zoom In (-)

Zoom in using the current factor.

Set Zoom Factor: Percentages

You may select a zoom factor here. The current one is checked.

Set Zoom Factor: None

When checked, this option sets the diagram size to its original scale and then inhibits further zooming until this option is unchecked.

Set Zoom Factor: Fit to Window

The diagram is stretched to fit the currently visible viewing area of the State Hierarchy View.

Highlight active states

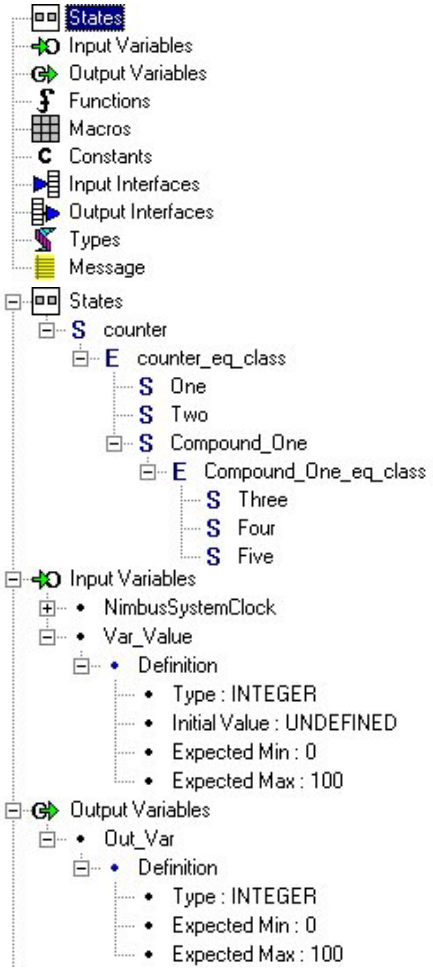
You may toggle the highlighting of active states. When this option is checked, active states are colored according to their fill color (a reconfigurable option from Color Options). Unchecking this item disables highlighting and is a useful feature for copying or saving structurally focused diagrams.

4.3.6 Tree View

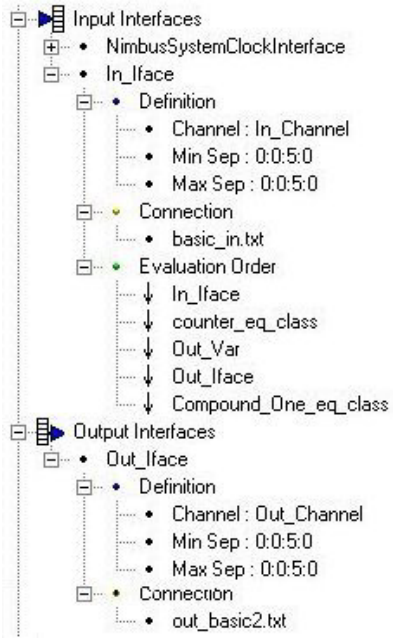
Specification details are contained in the Tree View. The first level of this view displays identifying names for key components in the system. The tree is refreshed when: a new specification is loaded, when the user presses F5, and when the user selects Refresh Views from the View Menu.

At the top of the tree, the state hierarchy is displayed where represents a State and represents an Equivalence Class.

Variables are displayed with information for Type, Expecting Min, Expected Max, and Initial Values (input variables only).



Interfaces are displayed with their constant attributes as well as the configurable Connection information. This is refreshed with new information when the user collapses then expands the connection or interface name nodes, or by pressing F5 or selecting Refresh Views from the View Menu.



4.3.7 State Hierarchy Diagram

The State Hierarchy Diagram is a visualization of the hierarchy of states. The user may configure its Color Options, Layout and Orientation Options, and other settings. You are free to select and then move, resize, or stretch the states and their connective links. All relationships to the specification are preserved.

Active States are colored as shown, with atomic states filled and non-atomics bordered. In the following example, the state *One* is active. Also, the user has selected state *Five* and is about to **Force State Active** via the popup menu. You will notice that **Force State Inactive** as well as other diagram options is accessible from this menu as well.

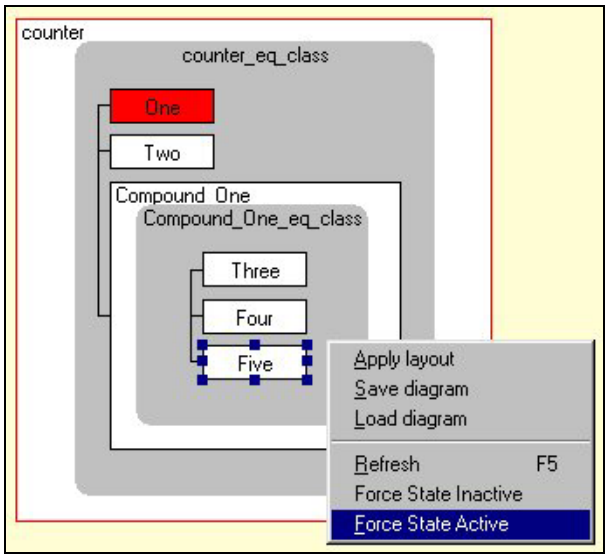


Figure 27 State Hierarchy Popup Menu: Forcing a State Active

4.3.8 Status Bar

The Status Bar contains information about the current system state and settings as shown.

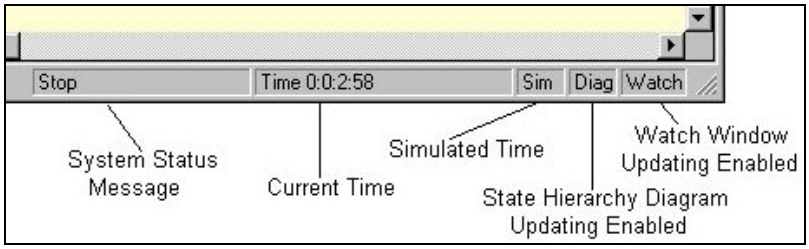


Figure 28 Status Bar

4.3.9 Stopping Conditions

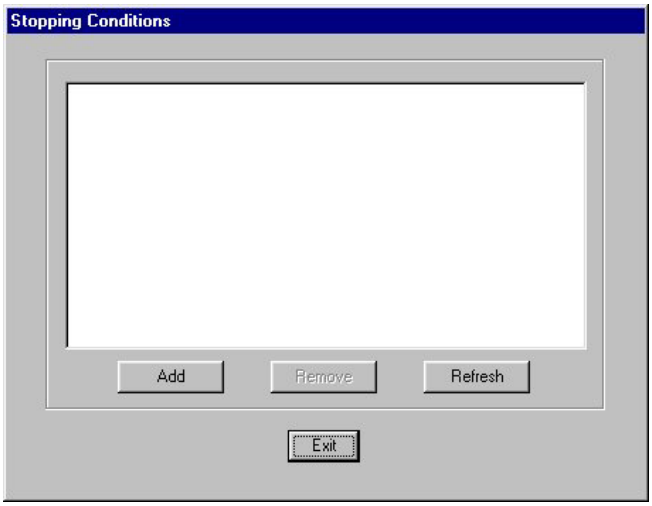


Figure 29 The Stopping Conditions Viewer

To add a stopping condition, push the **Add** button.

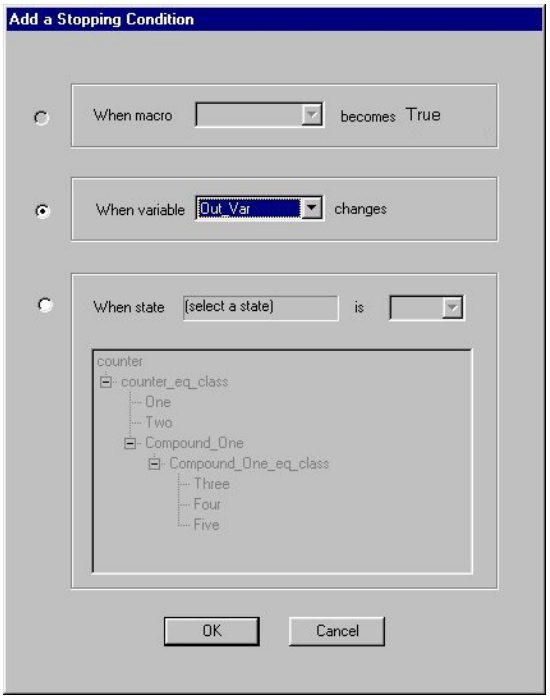


Figure 30 Stopping Conditions Example: Variable Change

In the next dialog, select the radio button for the item to which you want to add a stopping condition. In this example the user chooses to add the condition *When Out_Var changes*:

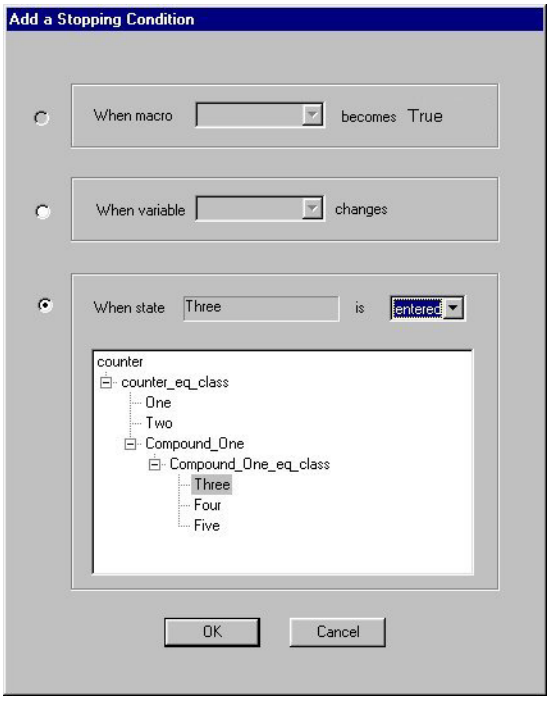


Figure 31 Stopping Conditions Example: State Entered

For states, select the item in the tree diagram and then choose the desired condition from the list. Here, the user adds the condition "*When state Three is entered"*.

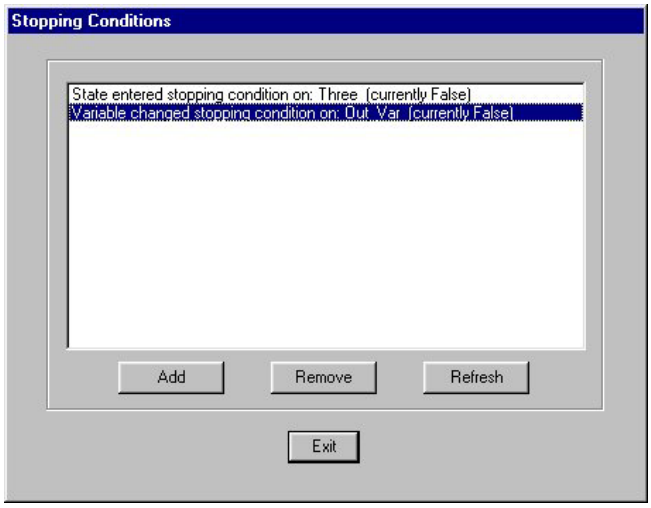


Figure 32 Stopping Conditions Viewer

The results are shown above. You may delete a condition: select it from the list and press the **Delete** button.

4.3.10 Watch Window

The Watch Window contains a listing of Variables and their values. When values change over time, the newly updated ones are colored red as shown.

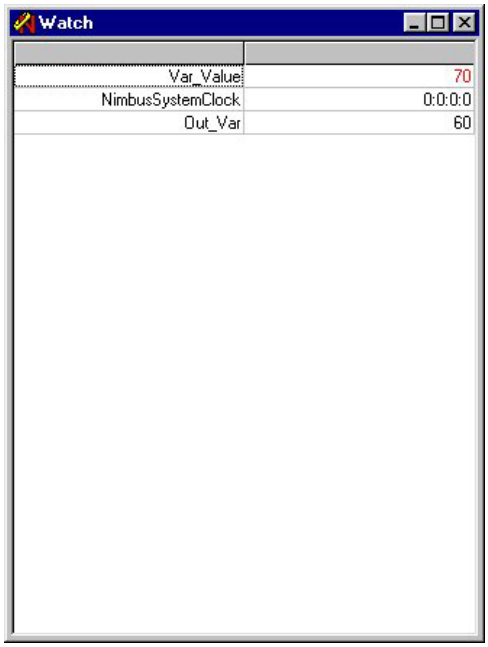


Figure 33 Watch Window

4.3.11 Channel Connector

To connect an interface to a channel:

- Select the interface from the list
- Choose the connection type **File** or **NIMBUS Channel**
- *Select the appropriate options*:
- For files, choose Single or Multiple and enter the source file name.
- For NIMBUS Channels that are Publish or Read, you have the option to select **Buffer**.

Here the user connects **In_IFace** to the file channel **basic_in.txt**:

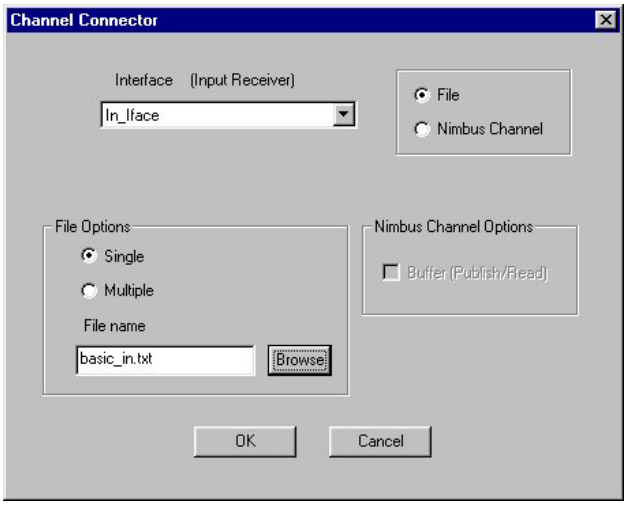


Figure 34 The Channel Connector

4.4 Running a Simulation

Here are a few quick steps that enable you to get simulation started.

Load a RSML^{-e} File

1. Load a **rsml** file from File menu or by clicking the icon to the left of the main toolbar
2. Connect Channels with the Channel Connector.
3. Add Stopping Conditions (optional)
4. Check/Set the Clock Type
5. Check/Set the Updating of the State Hierarchy Diagram and Watch Window
6. Bring the Watch Window to the foreground.
7. Run: Atomic, Microstep, Step, or Full Speed
8. Stop

Optionally, the user could have created a Script File of Command Line Interpreter directives and then loaded that file into the simulator.

Once a specification is loaded, you can visualize its structure with the Tree View and State Hierarchy Diagram.

As the execution proceeds, you can view its progress by observing: the highlighting of active states in the State Hierarchy Diagram, the changing of variable values in the Watch Window, the updating of the clock time in the Status Bar.

4.5 Command Reference

The following is a summary of commands that are accessible from the graphical user interface and the command line interpreter. There are some additional commands, not listed, that are unique to the GUI and the Command Line Interpreter. Use the help command to get the latest information on all commands available t the user.

Task	Graphical User Interface	Command Line
Help	[Help Menu | Main Toolbar]	[? | help]
Load a Specification	[File Menu | Main Toolbar | Ctrl+L]	load [filename]
View Specification	Tree View & State Hierarchy View	dump -s allStates
View Active States	State Hierarchy View (highlighted)	dump -s activeStates
View Input Interfaces	Tree View	dump -s inputHandlers
View Output Interfaces	Tree View	dump -s outputHandlers
View Input Variables	Tree View	dump -s inputVariables
View Output Variables	Tree View	dump -s outputVariables
Connect Channels	[Simulation Menu | Simulation Toolbar] Channel Connector	connect handler_options channel_options
View Stopping Conditions	[Simulation Menu | Simulation Toolbar] Stopping Conditions	dump -s stopConditions
Add Stopping Conditions	[Simulation Menu | Simulation Toolbar] Stopping Conditions Add	stopCondition s entityName
Remove Stopping Conditions	[Simulation Menu | Simulation Toolbar] Stopping Conditions Remove	stopCondition c [position]
Run One Atomic Step	[Simulation Menu | Simulation Toolbar | Shift+A]	run e
Run One Microstep	[Simulation Menu | Simulation Toolbar | Shift+M]	run m
Run One Step	[Simulation Menu | Simulation Toolbar | Shift+S]	run s
Run Full Speed	[Simulation Menu | Simulation Toolbar | Shift+F]	run f
Stop	[Simulation Menu | Simulation Toolbar | Shift+X]	stop
Force State Active	[Simulation Menu | Simulation Toolbar]	forceState [state_name_path|eq_class_name_path] -a
Force State Inactive	[Simulation Menu | Simulation Toolbar]	forceState [state_name_path|eq_class_name_path] -i
Set Time Model to Real-time	Simulation Menu: ✓**Realtime**	setoption driver realtime
Set Time Model to Sim-time	Simulation Menu: **Realtime**	setoption driver simtime
Load a Script File	[File Menu | Simulation Toolbar]	script [-read | -r] scriptFileName
State Hierarchy Layout	[State Hierarchy Menu | Toolbar]	

Options and Color Options		
State Hierarchy Copy, Load, Save, Zoom, Apply Layout	[State Hierarchy Menu | Toolbar]	
State Hierarchy Active State Highlighting	State Hierarchy Menu: ✓**Highlight Active State**	
Completeness and Consistency Analysis		ceAnalysis [subcommand
Backwards Execution Analysis		beAnalysis [subcommand]

5 Using NIMBUSChannel

5.1 Introduction

Specification and verification of software for safety critical systems is a difficult problem. There are three methods available to ensure the correctness of such systems: manual inspections, formal verification, and simulation and testing. All three approaches have various strengths and weaknesses; however, they are complementary. To achieve the high level of confidence in the correctness of the system necessary today's applications, all three approaches must be used in concert. The NIMBUSChannel communication framework supplies an important component of the simulation and testing aspect of the NIMBUS environment.

The NIMBUSChannel framework was developed to support the execution of RSML^{-e} models in conjunction with models of the other components in the environment. The framework was conceived in the work done at the University of Minnesota in the Critical Systems Research Group. During their work with RSML, the group developed a simulation that was capable of executing the formal definition of the language while taking input from text files. Nevertheless, the group discovered that these capabilities alone were not a sufficiently powerful environment in which to test a formal specification. It was prohibitively inflexible to create the text file input scripts in the early stages of execution because, most of the time, the user did not know exact sequences of input to test; they only wished to "debug" the formal specification. Furthermore, in cases where a dynamic model, for example, an aircraft, was involved, it was very difficult to create even semi-realistic input scripts for the simulation. These experiences led to the creation of the following requirements that any environment for the debugging and testing of formal specifications should support.

- It must support execution of the RSML^{-e} model while interacting with accurate models of the component's environment. These models should be able to be nearly anything the analyst might desire to model the component be that other RSML^{-e} specifications, software simulations, or even actual hardware in the environment.
- It must allow the analyst to easily modify and interchange the models of the components in the environment. Preferably, the components should be dynamically swappable at run time.
- As the specification is refined from high level requirements there should not be any large conceptual leaps in the way in which the control software communicates with the environment.

To meet these requirements, it was clear that some standard inter-process communication method would be necessary. Furthermore, the more high-level such a mechanism was, the better it would support the needs of NIMBUSChannel. Therefore, the CriSys group examined two technologies: COM (Component Object Model) and CORBA (Common Object Request Broker Architecture). These frameworks provide a high-level object-oriented communication interface, and allow transparent distributed object access. In the end, COM was selected because it was freely available for the target platform (Windows NT) and also supported by a greater number of

current applications and scripting technologies (i.e. Visual Basic for Applications and ActiveX scripting); thus, allowing greater flexibility at decreased cost.

The framework models interprocess communication as messages passing over simple channels. Currently, the only types supported by the channels are integer, enumerated, and boolean; in the future, the channels will most likely support floating point as well. Formally, all communication from a RSML^{-e} specification is one of two types: Send-Receive or Publish-Read. Send-Receive communication is a message passing protocol where the source pushes a message across the channel to the destination, causing an interrupt, or signal, in the receiving process. This type of communication is well suited to COM's function-call type of interaction. Publish-Read communication is where the source updates (or Publishes) the message on the channel intermittently and at any time the destination can pull the current message from the channel. This type of communication requires additional support because there must be a buffer (to hold the current message on the channel). There are a number of places that make logical sense for this buffer to reside.

- **Inside the publishing application.** In this model, when the destination of the channel wishes to read, the source of the channel is interrupted and produces the message requested. Storing the buffer here allows, for example, the destination of the channel to cause an update of a spreadsheet. Notice that in this model the publishing end of the channel never calls the "Publish" method of the channel-end component; instead, it simply waits to be called by the destination end of the channel. The subtype of the Publish-Read communication is known as PublishViaEvent-Read.
- **Inside the publishing channel component.** Alternatively, the buffer could reside inside the publishing end of the channel. Then, instead of passing it immediate over the channel, the channel-end component stores the message and waits to be called by the reader. When the reader asks for the message on the channel, the publishing channel-end simply passes the buffered message across the channel. This type is known as PublishWithBuffer-Read.
- **Inside the reading channel component.** Finally, the publishing end can immediately pass the message over the channel (like a Send-Receive channel) and the read channel, instead of interrupting the destination like in Send-Receive, stores the message in its buffer. Then, when the reading process need the message, the reader channel-end simply returns its buffered copy. This type is known as Publish-ReadWithBuffer.

These different options allow the user of NIMBUSChannel to optimize the number of inter-process COM calls and thereby increase the effective speed of the simulation environment. Furthermore, the ability to publish by event (i.e. interrupt the publishing process) allows for great flexibility in integrating applications like spreadsheets and databases into the NIMBUSChannel environment. More information about how these types of communication are implemented can be found in the Behind the Scenes section on page 106.

This document describes the NIMBUSChannel: its component applications, how to manage a simulation, how to create and add component models in the environment. To conserve space, this document assumes some familiarity with COM, C++, and Visual Basic. The examples given should be readable by any computer scientist; however, the reader should not expect to actually

be able to construct a NIMBUSChannel component without becoming familiar with the technologies involved.

5.2 Components of NIMBUSChannel

Aside from the NIMBUSChannel clients provided in the example applications, the NIMBUS Environment installation includes three main NIMBUSChannel client applications: The NIMBUSSim RSML^{-e} Simulator and Analysis tool, the NIMBUSChannel MFC client, and the NIMBUS Manager. These applications provide a basis from which to build a system simulation. A dynamic link library (DLL), type libraries, and a background executable process support them. This section describes each of these components and their use.

5.2.1 NIMBUSSim

NIMBUSSim is described in *The NIMBUSSim Graphical User Interface Manual* included in the NIMBUS environment documentation. To connect the interface of a RSML^{-e} specification to the NIMBUSChannel framework, simply select the interface you wish to connect in the channel connection dialog box and then choose NIMBUSChannel as the type of connection. For Publish or Read interfaces it is possible to select whether or not the buffer resides with the selected interface, or on the other end of the channel (see page 106, Behind the Scenes).

Figure 35 shows the channel connection dialog box. The example ASW specification has been loaded and the user is in the process of connecting the DOIStatusMessageInterface to the NIMBUSChannel environment. Because the DOIStatusMessageInterface specifies the receive type of communication, the Buffer option is not available and is grayed out.

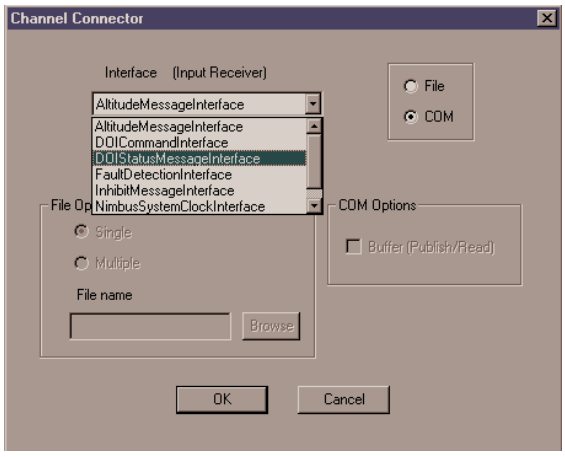


Figure 35: The channel connector dialog box in NimbusSim with COM selected

5.2.2 NIMBUSChannel Client

The NIMBUSChannel Client allows the user to send simple on-the-fly inputs to other components in the NIMBUSChannel architecture. Figure 36 shows the configuration dialog box for the client. The user specifies the Component name, the channel name, the type of communication, and the number of fields in the message that they are sending. The channel name is a string that identifies which channel in the NIMBUSChannel environment the client is connecting to. Channel names in the NIMBUSSim client are given in the specification document. The component name identifies the process (or component) supplying that end of the channel.

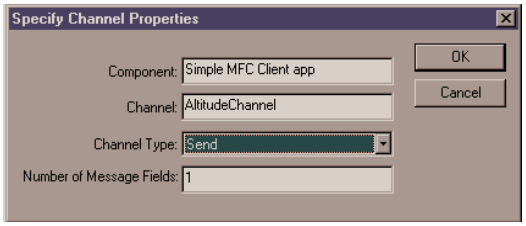


Figure 36: The configuration dialog for the simple MFC client

Once the user has selected the options necessary for channel configuration and pressed the OK button, the client creates a new channel-end object of the appropriate type and registers itself in the NIMBUSChannel framework. At that point, it is ready to participate in the framework and the main window of the client application appears.

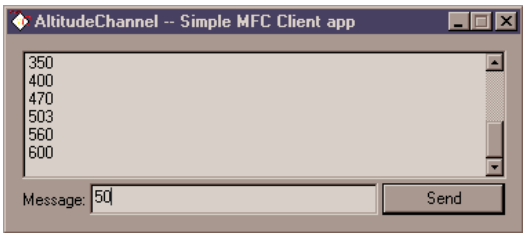


Figure 37: The simple MFC client connected to the AltitudeChannel

Figure 37 shows the main window of the MFC client application. The window supports all the various types of communication. The title bar shows the channel name on the left and the component name on the right so that the client's place in the framework can be easily identified.

5.2.3 NIMBUS Manager

The NIMBUS Manager allows the user to dynamically control the connections between the various components which are registered with the NIMBUSChannel system. Figure 38 shows the main window of the NIMBUS Manager. In the left-hand column, all the channels currently

registered in the system are listed. When the user clicks on a channel name, the detailed information about that channel is displayed in the area to the right of the channel list.

For each channel, the NIMBUS Manager displays the channel name (in Figure 38, AltitudeChannel), the type (Send-Receive) and three lists (sources, destinations, and observers). In the lists are the components that supply that channel. For example, Figure 38 shows that Altitude channel currently has two sources and two destinations. The sources are "MS Excel ASW System" from the example files and "Simple MFC Client app" which was instantiated in the previous section.

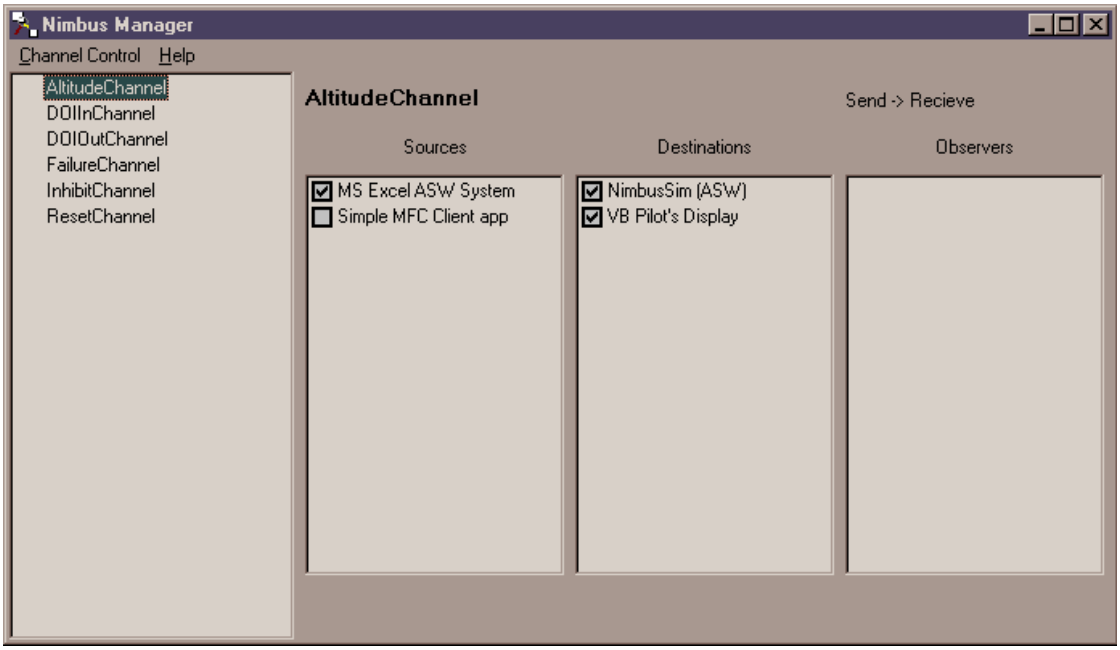


Figure 38: The main window of the NIMBUS Manager

The check marks beside the component names in Figure 38 indicate whether or not that component is currently active on the channel or not. A component that is not active cannot participate in the message processing (sending, receiving, publishing, etc) on that channel. Thus, Figure 38 shows that the "MS Excel ASW System" component is active, whereas the "Simple MFC Client app" is not. Notice in the figure that both "NIMBUSSim (ASW)" and "VB Pilot's Display" are active destinations on the AltitudeChannel. The NIMBUSChannel environment allows multicast communication to allow multiple different displays of the data. Also (not pictured in Figure 38) allowed are observers on a channel.

The NIMBUSChannel environment *does not* allow multiple sources on a channel. Thus, if the user where to click on the checkbox next to "Simple MFC Client app," then it would become the active source on the AltitudeChannel and "MS Excel ASW System" would become inactive. For convenience, the first registered source and destination on a channel are made active by default.

5.2.4 Behind the Scenes

The capabilities of NIMBUSChannel are implemented by a number of COM objects that implement several custom interfaces. This section explains briefly what a COM object is[1], what COM objects are used in NIMBUSChannel, and how the NIMBUSChannel objects are stored and implemented.

COM is an *executable* (binary) software component standard. Like Java, each COM object can support multiple interfaces. Each interface has a number of methods that can be called by the user of the object. Furthermore, to support dynamic run-time discovery of a component's capabilities, every COM interface inherits from an interface called IUnknown. The purpose of IUnknown is to allow the user to query for another interface on the object. If the query succeeds, then the client has a pointer to the requested interface. If not, the client receives an error message indicating that the interface as not supported.

COM is a binary standard. That is, as long as an object conforms to the binary specification it can be loaded into memory and run by the COM library. Therefore, COM is language independent. Currently, developers can create COM objects in C++, Visual Basic, or Java under a number of different development environments. The procedure for creating a COM object is to 1) write the code and compile it, and 2) register it with the COM library on the machine you wish it to run on. It is also possible to transparently call objects on different machines using DCOM.

Under Windows, COM objects are either packaged into and executable file, which runs on its own, or into a dynamic link library, which is loaded into a process. The NIMBUSChannel environment uses both techniques to package the various objects in the system. There are three types of objects in NIMBUSChannel

- **Channel Ends** are the objects that implement the details of the communication channels (i.e., implement Send-Receive, Publish-Read as discussed in the Introduction). There are a total of nine channel ends: Send, Receive, Publish, PublishWithBuffer, PublishWithEvent, Read, ReadWithBuffer, and, Observe. These objects shield the client processes from the details of the specific type of communication. They also handle all communication with the manager application.
- **Channel Wrapper** objects provide a convenient method of creating and destroying the channel end objects in the target development environment. Currently, there are channel wrapper objects available for both C++ and Visual Basic.
- The **Channel Manager** object manages the connection of channels in the system. This is a background process that is separate from the aforementioned NIMBUS Manager. The channel manager houses all the data structures displayed by the NIMBUS Manager and is the object which actually does the connecting and disconnecting of channels.

[1] Many references for COM are available. Among the best is *Inside COM* by Dale Rogerson (Microsoft Press).

The channel ends and channel wrapper objects are located in the RSMLChannelEnds.dll file whereas the channel manager is located in the RSMLChannelManager.exe file. Both these files are located in the distribution directory and are registered in the Windows registry.

5.3 Connecting Other Applications

Connecting other applications into the NIMBUSChannel framework boils down to getting those applications to create and use the channel wrapper and/or channel end objects discussed at the end of the previous section. There are several important questions to ask about the application that you wish to integrate:

- Does the application have an API or macro capability that would allow other applications access to its internal data structures? If not, do you have access to the source code of the application?
- Does the application support ActiveX scripting technologies, for example, Microsoft's Visual Basic for Applications.
- Does the application support OLE/COM automation.

The ease of integrating any application into NIMBUSChannel is dependent upon how full featured the API of that application is. Of course, if the application does not provide any API and there is not access to the source of the application, it will be impossible to easily integrate it into the NIMBUSChannel framework. If there is an API accessible through, for example, C or C++ then a NIMBUSChannel C++ Client might be the way to go unless a better alternative is available.

If the application supports OLE automation, then the application supports a set of COM interfaces that allow Visual Basic (or any other application) to cause various commands of the application to be run. Essentially, it allows for remote control of the application. Thus, the control the application and the channels can be written in one Visual Basic application and the developer can use the VB wrapper.

The easiest and fastest way to achieve integration is to use an application that supports ActiveX scripting ActiveX scripting is a standard where applications support the creating and use of COM objects within the context of a built-in scripting or macro language. Visual Basic for Applications (VBA) is one common example of an ActiveX scripting language. Using a language like VBA is absolutely the fastest way to integrate a new application into the system and it allows the most flexibility.

The general procedure that is followed in creating clients for NIMBUSChannel is the following.

1. Declare a variable to hold the instance of the channel wrapper object.
2. Create the channel wrapper object.
3. Configure the channel wrapper object by filling in the channel name, component name, and channel type properties.
4. Call the register method to have your new channel join the NIMBUSChannel environment.

5. Use the channel.
6. Call the unregister method when you are through using the channel.

The following sections describe the creating of a visual basic and a C++ Client. The focus is on the code which is necessary to make the application work in the NIMBUSChannel framework, not on the functionality of the application. Thus, much code that deals with, for example, the user interface of the application is omitted.

5.3.1 Building a Visual Basic Client

Visual basic clients (including VBA clients) are the easiest to make. The example in this section shows the creation of a client that has two channels: sending and receiving. The example omits any of the user interface code normally associated with a VB application and instead focuses on the code necessary to make the channels function.

Below, you see the first step in creating a NIMBUSChannel client: the variables for the objects have been declared.

```
Dim WithEvents MySendChannel As RSMLChannelVBWrapper
Dim WithEvents MyReceiveChannel As RSMLChannelVBWrapper
```

These variables are declared "WithEvents" because for the receive, observe, and publish via event channel types, the channel end objects will cause an event to occur in Visual Basic. Note, that the caller (i.e., the sender) ***is blocked*** until the event procedure involked by the receive event is finished executing. Therefore, it is highly undesirable to make additional NIMBUSChannel calls within the event procedures as this would most likely cause a deadlock.

Object variables in Visual Basic are not initialized automatically to new objects. Thus, somewhere in the code, it is necessary to set the MySendChannel variable to a new instance of the RSMLChannelVBWrapper object. After the object is created, the programmer fills in the properties necessary to initialize the object: the channel name, component name, and channel type. Finally, the object is registered with the channel manager by calling the "Register" method. This code is shown on the following page.

```
Sub InitChannels()

   '
   ' MySendChannel
   '
   Set MySendChannel = new RSMLChannelVBWrapper

   With MySendChannel
      .ChannelName = "SendingChannel"
      .ChannelType = rsmlSend
      .ComponentName = "My First VB Client"
   End With

   MySendChannel.Register

   '
   ' MyReceiveChannel
```

```
    '
    Set MyReceiveChannel = new RSMLChannelVBWrapper

    With MyReceiveChannel
       .ChannelName = "ReceivingChannel"
       .ChannelType = rsmlReceive
       .ComponentName = "My First VB Client"
    End With

    MyReceiveChannel.Register

End Sub
```

After the InitChannels function (above) has completed, both the sending and receiving channel clients will be ready for use. For the sending channel, we can make a VB interface form to allow the user to send the message (not shown). However, for the receiving VB will call the event procedure pictured below.

```
Sub MySendChannel_receive(msg() As Long)
    '
    ' Display Message shows the message to the user
    '
    DisplayMessage msg
End Sub
```

The procedure takes one argument: an array of long integers. This array is the message that is passed over the channel. Enumerated types and Boolean values are converted to integers (0 for false, 1 for true, 0 indexed for the enumerated types) before they are passed over the channel. The developer can put any code that he or she likes in the event procedure; however, note again that ***the sender is blocked until the call returns***.

```
Sub CleanupChannels()
    MySendChannel.Unregister
    MyReceiveChannel.Unregister
End Sub
```

Finally, after the application is done using the NIMBUSChannel framework it should unregister the channel end objects so that the channel manager can disconnect and remove them from the system.

5.3.2 Building a C++ Client

Creating a C++ Client is much like creating a Visual Basic client. The differences many lie in syntax and the fact that in C++ it is necessary to deal with VB compliant types, for example, the SAFEARRAY structure that is used to pass the messages over the channels. These topics are beyond the scope of this guide, and, in practice, creating a custom C++ client is a rare occurrence under Windows. An example of a C++ Client, written as a dialog based MFC application, is the simple channel client distributed with the NIMBUS environment. The source for this client can be obtained from Jeff Thompson (thompson@cs.umn.edu).

5.4 Troubleshooting and Common Issues

This section is under development and will evolve as the tools get more use.

6 References

[Berry00] G. Berry. The Foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner,* MIT Press, 2000

[Halbwachs91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Dataflow Programming Language Lustre. In *Proceedings of the IEEE*, Volume 79, #9, pp. 1305-20, September 1991.

Appendix A - Using the Airlock Interface

The VB Airlock interface is used to provide the environment for an airlock control specification. It uses the NIMBUS-Channel communications framework to communicate with a specification. The VB program periodically sends a message to the specification describing the status of the environment and receives a message with actuator commands from the specification. The input message for the specification is sent over the channel named "inputChannel", and the VB client receives the output message from the specification over the channel named "outputChannel". By connecting an RSML specification to these channels, it is possible to communicate with the VB client.

A.1 Message Specifications

The message specifications are as follows:

```
MESSAGE Environment_Message {
  f_panic_button IS boolean,
  f_reset_button IS boolean,
  f_inner_door_request IS boolean,
  f_outer_door_request IS boolean,
  f_inner_door_open IS boolean,
  f_outer_door_open IS boolean,
  f_airlock_occupied IS boolean
}
```

This message is received by the specification, and describes the status of, respectively:

- the panic alarm button
- the reset button
- the inner door request button
- the outer door request button
- whether or not the inner door is open
- whether or not the outer door is open
- whether or not the airlock is occupied

From this information, it should be possible to create a specification controlling the behavior of the airlock.

The message sent to control the actuators of the airlock is as follows:

```
MESSAGE Actuator_Message {
  f_inner_door_lock IS boolean,
  f_outer_door_lock IS boolean,
  f_decontaminate IS boolean,
  f_panic_alarm IS boolean,
  f_timeout_alarm IS boolean
}
```

This message describes actuator commands to, respectively:

- lock (TRUE) or unlock (FALSE) the inner door
- lock (TRUE) or unlock (FALSE) the outer door
- decontaminate the chamber
- turn on/off the panic alarm
- turn on/off the timeout alarm.

These actuators control the operation of the airlock.

A.2 Creating Specification Interfaces.

In order to connect the specification to the VB Airlock, it is first necessary to create interfaces to receive/send messages. First, we need an input interface to receive in the environment message. Here is some sample code that will do this step:

```
MESSAGE Environment_Message {
   f_panic_button IS boolean,
   f_reset_button IS boolean,
   f_inner_door_request IS boolean,
   f_outer_door_request IS boolean,
   f_inner_door_open IS boolean,
   f_outer_door_open IS boolean,
   f_airlock_occupied IS boolean
}

IN_VARIABLE panic_button : boolean
   INITIAL_VALUE : FALSE
   CLASSIFICATION: MONITORED
END IN_VARIABLE

IN_VARIABLE reset_button : boolean
   INITIAL_VALUE : FALSE
   CLASSIFICATION: MONITORED
END IN_VARIABLE
```

```
IN_VARIABLE inner_door_request : boolean
   INITIAL_VALUE : FALSE
   CLASSIFICATION: MONITORED
END IN_VARIABLE

IN_VARIABLE outer_door_request : boolean
   INITIAL_VALUE : FALSE
   CLASSIFICATION: MONITORED
END IN_VARIABLE

IN_VARIABLE inner_door_open : boolean
   INITIAL_VALUE : FALSE
   CLASSIFICATION: MONITORED
END IN_VARIABLE

IN_VARIABLE outer_door_open : boolean
   INITIAL_VALUE : FALSE
   CLASSIFICATION: MONITORED
END IN_VARIABLE

IN_VARIABLE airlock_occupied : boolean
   INITIAL_VALUE : FALSE
   CLASSIFICATION: MONITORED
END IN_VARIABLE

IN_INTERFACE Update_Interface :
       MIN_SEP : UNDEFINED
       MAX_SEP : UNDEFINED
       INPUT_ACTION : RECEIVE(Environment_Message)
       HANDLER :
              CONDITION : TRUE
       ASSIGNMENT
         panic_button         := f_panic_button,
         reset_button         := f_reset_button,
         inner_door_request   := f_inner_door_request,
         outer_door_request   := f_outer_door_request,
         inner_door_open      := f_inner_door_open,
         outer_door_open      := f_outer_door_open,
         airlock_occupied     := f_airlock_occupied
             END ASSIGNMENT
       END HANDLER
END IN_INTERFACE
```

Similarly, the output interface must be connected:

```
MESSAGE Actuator_Message {
   f_inner_door_lock IS door_lock_status,
   f_outer_door_lock IS door_lock_status,
   f_decontaminate IS boolean,
   f_panic_alarm IS on_off,
   f_timeout_alarm IS on_off
}
```

```
OUT_INTERFACE Actuator_Interface :
       MIN_SEP : UNDEFINED
       MAX_SEP : UNDEFINED
       OUTPUT_ACTION : SEND(Actuator_Message)
       HANDLER :
              CONDITION : TRUE
              ASSIGNMENT
         f_inner_door_lock    := inner_door_lock,
         f_outer_door_lock    := outer_door_lock,
         f_decontaminate      := decontaminate,
         f_panic_alarm        := panic_alarm,
         f_timeout_alarm      := timeout_alarm
              END ASSIGNMENT
      ACTION : SEND
       END HANDLER
END OUT_INTERFACE
```

where `inner_door_lock`, etc., should be state variables within your specification.

A.3 Connecting the RSML Channels

In order to "wire together" the VB client with the RSML specification, you must connect the RSML interfaces to the VB interfaces. On the VB side, this is embedded in the application; however, we must manually connect the interfaces on the RSML side. You can either connect the interfaces from the RSML command line, or you can write a script file, which will connect the interfaces whenever the specification loads, which is the recommended procedure.

Creating a Script File

To create an automatically loading script file, create a text file with the name <SPECIFICATION_NAME>.nscript, where SPECIFICATION_NAME is the name of the RSML specification. If this file is in the same directory as the specification, it will be automatically loaded when the specification is loaded.

Script File Contents

Here is an example script file that connects the Update_Interface and the Actuator_Interface (described above) to the channels used by the Visual Basic client. The name of the RSML specification that uses this file is Cleanroom.nimbus.

Cleanroom.nscript:

```
connect -i Update_Interface inputChannel -com
connect -o Actuator_Interface outputChannel -com
setoption driver realtime
```

The *connect* command connects an interface to a communications channel. The *–i* option is used to specify that this connection should be for *input* into the specification. Similarly, *–o* signifies output from the specification. Next are the names of the interface and channel to be connected,

respectively. The final argument specifies that this channel is a COM channel. Nimbus can read/write over several channel types: for example, it is possible to create input files and use them as channels. COM channels use Microsoft's COM to communicate in real time between applications.

The *setoption* command is used to toggle several simulation options. In this case, it sets the simulator to run in *realtime* mode, as opposed to *simulated time* mode. In *simulated time* mode, the specification runs much faster than real time. This mode is useful when replaying a log of generated events or using file channels. However, for COM channels, the specification must be run in real time mode for correct results.

A.4 Executing the Nimbus Simulator and the VB Driver

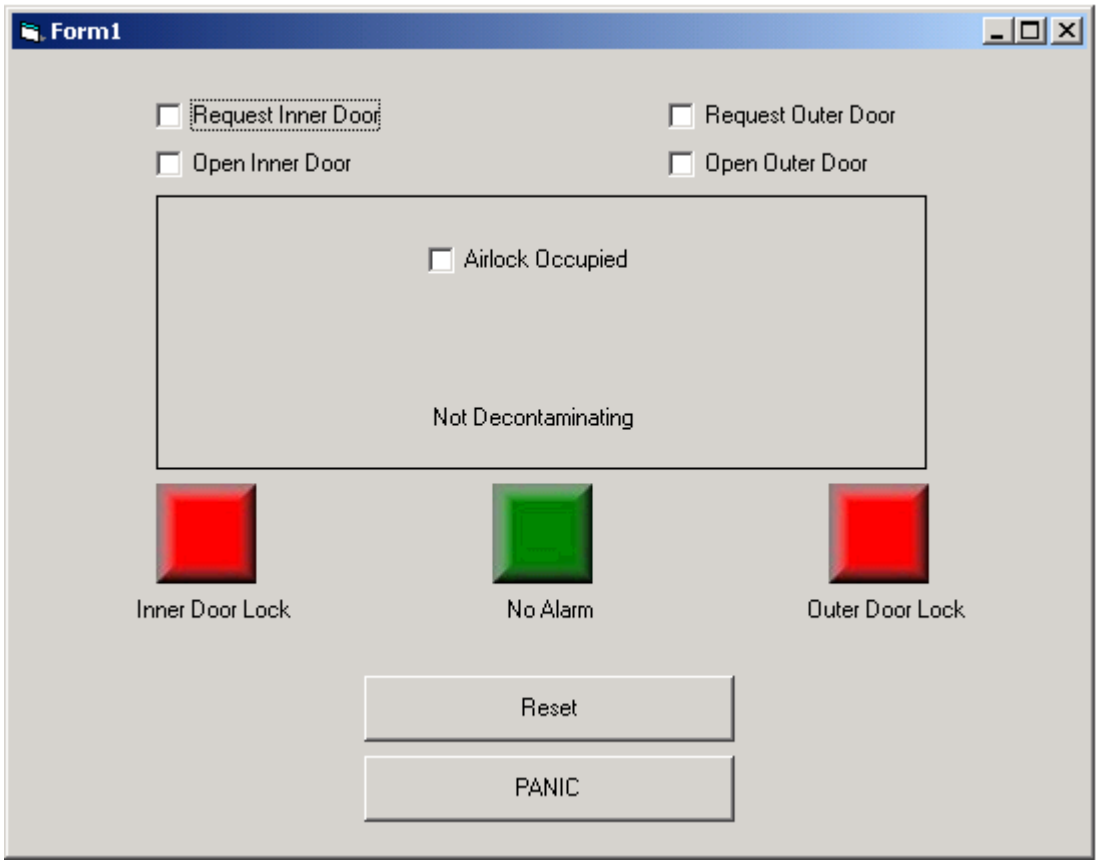
Once a script file has been created, it is straightforward to simulate the behavior of the specification. First, start the VB Driver executable. Then, start the RSML simulator, and load the specification. At this point, if your script file is correctly written, all channels should be wired together. You can check by looking at the Nimbus command line window after loading the specification; it should read something like:

```
Script file name: C:\data\srcsafe\Members\whalen\csci8990\cleanroom.nscript
executing command: connect -i Update_Interface inputChannel -com
Finding channel manager.
Registering with channel manager.
executing command: connect -o Actuator_Interface outputChannel -com
Finding channel manager.
Registering with channel manager.
executing command:
executing command: setoption driver realtime
executing command:
completed script file C:\data\srcsafe\Members\whalen\csci8990\cleanroom.nscript
5 lines processed.
```

If you don't see these lines in the command line window, it is likely that the file name of the script file is wrong; make sure that the file does not have a .txt extension on the end.

Finally, you should be able to simply start the simulator and test your work.

The VB Interface looks like this:



When the checkboxes are checked, the "actions" are occurring. So by checking "Open Inner Door" (if the door is unlocked), the user has opened the inner door. These conditions continue to hold until the box is unchecked.

The buttons at the bottom (reset and PANIC), simulate the user pressing either the reset or the panic button. In this implementation, these activities are only true for one instant and then false thereafter.

The status lights determine the state of the system. Red and Green mean *locked* and *unlocked*, respectively, for the doors. Similarly, Green means *no alarm*, and red means *alarm on*.

That's about all there is to it. Have fun & let us know of any errors.

Appendix B - Textual Grammar of RSML^{-e}

The full grammar for RSML^{-e} is given below.

```
component_def           : def_list
                        ;

def_list                : /* empty */
                        | def_list def
                        ;

def                     : type_def
                        | constant_def
                        | state_variable_def
                        | in_variable_def
                        | in_interface_def
                        | out_interface_def
                        | macro_def
                        | function_def
                        | message_def
                        ;


/*----------- State definition rules -------------------------------*/

state_variable_def      : STATE_VARIABLE IDENTIFIER array_decl ':'
variable_type_decl
                            PARENT        ':' parent_decl
                            INITIAL_VALUE ':' expression  /* checked to be
const */
                            variable_numeric_decl
                            classification_def
                            case_list
                          END STATE_VARIABLE
                        ;

variable_numeric_decl   : /* empty */
                        | UNITS         ':' IDENTIFIER   /* added to variable
properties */
                          EXPECTED_MIN  ':' expression   /* checked to be
const */
                          EXPECTED_MAX  ':' expression   /* checked to be
const */

variable_type_decl      : type_ref
                        | VALUES    ':' '{' enum_element_list '}'
```

```
array_decl              : /* empty */
                        | '[' expression TO expression ']' /* both
expressions checked to be const */
                        ;

parent_decl             : NONE
                        | parent_name_path
                        ;

parent_name_path        : IDENTIFIER
                        | parent_name_path '.' IDENTIFIER
                        ;

classification_def      : /* empty */
                        | CLASSIFICATION ':' IDENTIFIER
                        ;



/*------------ Type definitions, Only for enumerated types ---------*/

/* The type definitions can be safely ignored, as they should have
   been handled in the first pass. */

type_def                : TYPE_DEF IDENTIFIER '{' enum_element_list '}'
                        ;

enum_element_list       : IDENTIFIER
                        | enum_element_list ',' IDENTIFIER
                        ;

type_ref                : IDENTIFIER
                        | INTEGER_TYPE
                        | REAL_TYPE
                        | BOOLEAN_TYPE
                        | TIME
                        ;


/*------------ Message definition rules ---------------------------*/

message_def             : MESSAGE IDENTIFIER '{' field_list '}'
                        ;

field_list              : /* empty */
                        | IDENTIFIER IS type_ref
                        | field_list ',' IDENTIFIER IS type_ref
                        ;

/*------------ Constant definition rules --------------------------*/

constant_def            : CONSTANT IDENTIFIER ':' type_ref
                              UNITS ':' IDENTIFIER
                              VALUE ':' expression /* checked to be const */
                            END CONSTANT
```

```
                        | CONSTANT IDENTIFIER ':' type_ref
                            VALUE ':' expression /* checked to be const */
                          END CONSTANT
                        ;

/*------------ Variable definition rules ---------------------------*/


in_variable_def         : IN_VARIABLE IDENTIFIER
                            array_decl ':' type_ref
                            INITIAL_VALUE ':' expression /* checked to be
const */
                            variable_numeric_decl
                            classification_def
                          END IN_VARIABLE
                        ;


/*------------- Input Interface definitions -------------------------------
*/


in_interface_def        : IN_INTERFACE IDENTIFIER ':'
                            MIN_SEP ':' expression     /* checked to be
const */
                            MAX_SEP ':' expression     /* checked to be
const */
                            INPUT_ACTION ':' in_interface_type_spec '('
IDENTIFIER ')'
                            in_handler_list
                          END IN_INTERFACE
                        ;

in_interface_type_spec  : RECEIVE
                        | READ
                        ;

in_handler_list         : in_handler
                        | in_handler in_handler_list
                        ;

in_handler              : in_handler_type ':'
                            CONDITION ':' condition
                            in_assignment
                          END HANDLER
                        ;

in_handler_type         : RECEIVE_HANDLER
                        | HANDLER
                        ;

in_assignment           : /* empty */
                        | ASSIGNMENT
                           in_assignment_list
                          END ASSIGNMENT
                        ;
```

```
in_assignment_list      : identifier_name_path ASSIGN_TOKEN expression
                        | in_assignment_list ',' identifier_name_path
ASSIGN_TOKEN expression
                        ;


/*------------- Output Interface definitions -------------------------------
*/

out_interface_def       : OUT_INTERFACE IDENTIFIER ':'
                              MIN_SEP ':' expression     /* checked to be
const */
                              MAX_SEP ':' expression     /* checked to be
const */
                              OUTPUT_ACTION ':' out_interface_type_spec '('
IDENTIFIER ')'
                              output_handler_list
                          END OUT_INTERFACE
                        ;

out_interface_type_spec : SEND
                        | PUBLISH
                        ;

output_handler_list     : output_handler
                        | output_handler_list output_handler
                        ;

output_handler          : HANDLER ':'
                              CONDITION ':' condition
                              out_assignment
                              ACTION ':' out_handler_type
                          END HANDLER
                        ;

out_handler_type        : SEND
                        | PUBLISH
                        | NONE

out_assignment          : /* empty */
                        | ASSIGNMENT
                           out_assignment_list
                          END ASSIGNMENT
                        ;

out_assignment_list     : IDENTIFIER ASSIGN_TOKEN expression
                        | IDENTIFIER ASSIGN_TOKEN expression ','
                          out_assignment_list
                        ;


/*------------ Macro definitions -----------------------------------*/

optional_formal_parms   : /* EMPTY */
                        | '(' formal_parameter_list ')'
```

```
macro_def               : MACRO IDENTIFIER optional_formal_parms ':'
                              condition
                          END MACRO
                        ;


/*------------ Function definitions ---------------------------------*/
optional_expr_list      : /* empty */
                        | expression_list

function_def            : FUNCTION IDENTIFIER '(' formal_parameter_list ')'
':' type_ref
                              case_list
                          END FUNCTION

                        | STUB_FUNCTION IDENTIFIER '(' formal_parameter_list
')' ':' type_ref
                              optional_expr_list
                          END STUB_FUNCTION
                        ;

case_list               : /* EMPTY */
                        | case_list case
                        ;

case                    :  EQUALS expression IF condition
                        | TRANSITION expression TO expression IF condition
                        ;

actual_parameter_list   : /* empty */
                        | expression_list
                        ;

/* formal parameter lists are set up in the first pass */
formal_parameter_list   : /* empty */
                        | IDENTIFIER IS type_ref
                        | formal_parameter_list ',' IDENTIFIER IS type_ref
                        ;


/*-------- Identifier path definition rules -----------------------------*/


/*
   for the identifier_name_path rule, we don't need to check whether
   an array_ref is NULL, because the rule will work correctly in either
   case.  Specifiers can leave array_ref blank in the case of a non-array
   variable or if they wish to use an implicit 'this' expression.

      We also assume that IDENTIFIER will always return non-NULL values.
*/

identifier_name_path    : IDENTIFIER array_ref
                        | '.' '.' IDENTIFIER array_ref
```

```
                        | identifier_name_path '.' IDENTIFIER array_ref
                        | identifier_name_path '.' '.' IDENTIFIER array_ref
                        ;

array_ref               : /* empty */
                        | '[' expression ']'
                        ;


/*---------- Definitions defining AND/OR tables ----------------------*/

condition               :  TABLE
                           row_list
                           END TABLE
                        |  expression /* Must return BOOLEAN */
                        ;

row_list                :  expression ':' truth_value_list ';'
                        |  row_list  expression ':' truth_value_list ';'
                        ;

truth_value_list        :  truth_value
                        |  truth_value truth_value_list
                        ;

truth_value             : 'T'
                        | 'F'
                        | '.'
                        | '*'


/*---------- Expression definition rules ---------*/

expression              : unary_expression
                        | binary_expression
                        | array_expression
                        | event_expression
                        | time_expression
                        | prev_expression
                        | '(' expression ')'
                        | ASSIGNED '(' identifier_name_path ')'
                        | CHANGED '(' identifier_name_path ')'
                        | IDENTIFIER '(' actual_parameter_list ')'
                        | literal
                        | expression EQ_ONE_OF '{' expression_list '}'

unary_expression        : '-' expression %prec UMINUS
                        | NOT expression

binary_expression       : expression '*' expression
                        | expression '/' expression
                        | expression '+' expression
                        | expression '-' expression
                        | expression '>' expression
```

```
                        | expression '<' expression
                        | expression LESS_OR_EQUAL expression
                        | expression GREATER_OR_EQUAL expression
                        | expression EQUAL expression
                        | expression NOT_EQUAL expression
                        ;

array_expression        : EXISTS '(' IDENTIFIER ','

      identifier_name_path ',' expression ')'
                        | FORALL '(' IDENTIFIER ','

      identifier_name_path ',' expression ')'
                        | COUNT '(' IDENTIFIER ','

      identifier_name_path ',' expression ')'
                        | FIRST_INDEX  '(' IDENTIFIER ','

      identifier_name_path ',' expression ')'
                        | LAST_INDEX  '(' IDENTIFIER ','

      identifier_name_path ',' expression ')'
                        ;

event_expression        : AT_TRUE '(' expression ')'
                        | AT_FALSE '(' expression ')'
                        | AT_CHANGED '(' expression ')'
                        ;

prev_step_expression    : identifier_expression
                        | PREV_STEP   '(' identifier_expression ')'
                        ;

optional_pv             : /* EMPTY */
                        | ',' INT_VALUE
                        ;

optional_ta             : /* EMPTY */
                        | ',' INT_VALUE
                        ;

prev_expression         : prev_step_expression
                        | PREV_ASSIGN '(' prev_step_expression optional_pv
')'
                        | PREV_VALUE  '(' prev_step_expression optional_pv
')'
                        | TIME_ASSIGNED
                          '(' prev_step_expression optional_ta ')'
                        | TIME_CHANGED
                          '(' prev_step_expression optional_ta ')'
                        ;


static_variable_info_decl     : EXPECTED_MIN
                                                  | EXPECTED_MAX
                                                | UPPER_BOUND
```

```
                                                | LOWER_BOUND
                                                   ;

identifier_expression   : identifier_name_path
                        | identifier_name_path DOUBLE_COLON
static_variable_info_decl
                        | identifier_name_path DOUBLE_COLON MAX_SEP
                        | identifier_name_path DOUBLE_COLON MIN_SEP
                        | identifier_name_path DOUBLE_COLON THIS
                        | identifier_name_path DOUBLE_COLON TIME
                        | identifier_name_path DOUBLE_COLON LAST_IO

time_expression         : TIME
                        | HOURS '(' expression ')'
                        | MINUTES '(' expression ')'
                        | SECONDS '(' expression ')'
                        | MILLISECONDS '(' expression ')'
                        ;

expression_list         : expression
                        | expression_list ',' expression
                        ;

literal                 : time_literal
                        | INT_VALUE
                        | REAL_VALUE
                        | TRUE_TOKEN
                        | FALSE_TOKEN
                        | UNDEFINED
                        ;


time_literal            : tl_comp_list
                        ;

tl_comp_list            : INT_VALUE tl_units
                        | tl_comp_list tl_separator INT_VALUE tl_units
                        ;

tl_separator            : /* empty */
                        | AND
                        ;

tl_units                : H_TOKEN
                        | MIN_TOKEN
                        | S_TOKEN
                        | MS_TOKEN
                        ;
```